Approved for public release; distribution is unlimited

SCHEDULING TECHNIQUES FOR MULTIPLE PROCESSOR SYSTEMS IN REAL-TIME ENVIRONMENTS

by

John Howard Quigg
Captain, United States Army
B.S., United States Military Academy, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1993

Ted Lewis, Chairman,
Department of Computer Science

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Master's Thesis, July 1991 - September 1993

4. TITLE AND SUBTITLE
Scheduling Techniques for Multiple Processor Systems in Real-Time Environments(U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Quigg, John Howard

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-500

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Directed Acyclic Graph Scheduling is a technique used to implement the real-time execution of Digital Signal Processing applications on multiple-processor data-flow machines that support variable-grained parallelism. The approach used in the Navy's AN/UYS-2 Digital Signal Processor statically schedules an application graph at run-time using a First-Come-First-Served (FCFS) policy. Research by Shukla and Zaky [Shukla 91] developed a new algorithm, the Revolving Cylinder(RC), to ameliorate the inherently non-deterministic output flow of the FCFS scheduling approach currently used in the system.

Although the RC technique solved the problem of output-flow determinism there was no broad coverage of other current research in the very specialized field of real-time data-flow machines.

This thesis reviews Revolving Cylinder analysis and then surveys, compares, and evaluates research in the field using the review as a baseline for comparison. The RC approach is best at improving the throughput and output flow determinism of a narrow range of applications on a particular architecture. Each of the other approaches offer improvements over RC scheduling in either performance as measured by throughput or through flexibility in applications handled. For each of these improvements, however, significant trade-offs are made and so improvements become relative when they affect system robustness and an ability to handle repeated execution of application graphs. The AN/UYS-2 can implement RC scheduling with a minimum of cost and no hardware reconfiguration and this makes it the best approach for short-term system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Data-Flow, Digital Signal Processing, Real-Time, Graph Restructuring, Throughput	77
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

ABSTRACT

Directed Acyclic Graph Scheduling is a technique used to implement the real-time execution of Digital Signal Processing applications on multiple-processor data-flow machines that support variable-grained parallelism. The approach used in the Navy's AN/UYS-2 Digital Signal Processor statically schedules an application graph at run-time using a First-Come-First-Served (FCFS) policy. Research by Shukla and Zaky [Shukla 91] developed a new algorithm, the Revolving Cylinder(RC), to ameliorate the inherently non-deterministic output flow of the FCFS scheduling approach currently used in the system.

Although the RC technique solved the problem of output-flow determinism there was no broad coverage of other current research in the very specialized field of real-time data-flow machines.

This thesis reviews Revolving Cylinder analysis and then surveys, compares, and evaluates research in the field using the review as a baseline for comparison. The RC approach is best at improving the throughput and output flow determinism of a narrow range of applications on a particular architecture. Each of the other approaches offer improvements over RC scheduling in either performance as measured by throughput or through flexibility in applications handled. For each of these improvements, however, significant trade-offs are made and so improvements become relative when they affect system robustness and an ability to handle repeated execution of application graphs. The AN/UYS-2 can implement RC scheduling with a minimum of cost and no hardware reconfiguration and this makes it the best approach for short-term system improvement.

TABLE OF CONTENTS

I. INTRODUCTION

Today's military battlefield is one of ever increasing lethality. Tomorrow's combatants must have the ability to respond to threats within milliseconds to ensure their survival. These narrowing reaction windows necessitate both accurate and timely responses. Real-time processors ensure that these responses are performed within a known, guaranteed bound or deadline. This allows the designer of an application to use that bound with confidence that the system will return a result swiftly and reliably. Examples of real-time systems currently in use are those in aircraft cockpits, weapon sensors, and navigation systems. All of these handle increasingly complex tasks at high data rates and must do so without failure.

The robustness of these systems is vital because of the tremendous penalty for failure. Most real-time systems are embedded in some larger system and must have a high degree of fault tolerance to ensure the survivability of the platform [Levine 91]. Many of today's real-time systems have multiprocessor based architectures which increase throughput by sharing workloads. This facilitates graceful degradation in the event of failure by having multiple instances of each resource amongst which to spread a load.

A. Digital Signal Processing

Digital Signal Processing (DSP) is one of the applications standing to benefit by a departure from von Neumann style architectures. It is widespread and of particular use to the military on platforms ranging from submarines to spacecraft. DSP applications are well suited for description using Large Grain Data-flow Graphs (LGDF) because they can be described using a combination of mathematical expressions and block diagrams. The data-flow paradigm preserves the integrity of the flow of data and as a result allows the natural exploitation of any concurrency in the graph [Lee 87].

B. The AN/UYS-2

In the 1980's the Navy realized the potential of data-flow architectures and developed the AN/UYS family of DSP's. The AN/UYS-2, the system with which we are most concerned, was developed in order to introduce a standard DSP for military land, sea, and air applications. It is a variable-configuration multiprocessor based on the use of Standard Electronic Modules or SEM's. There are two different SEM's available: Type B and type E. The type E modules perform the same functions as those of type B but they are smaller, lighter, and more power efficient. They were developed for aircraft use because of the limitations imposed by limited space/lift in an airframe.

The modules are built from off-the-shelf hardware and are used to construct the processor's Functional Elements (FE) [Rice 90].

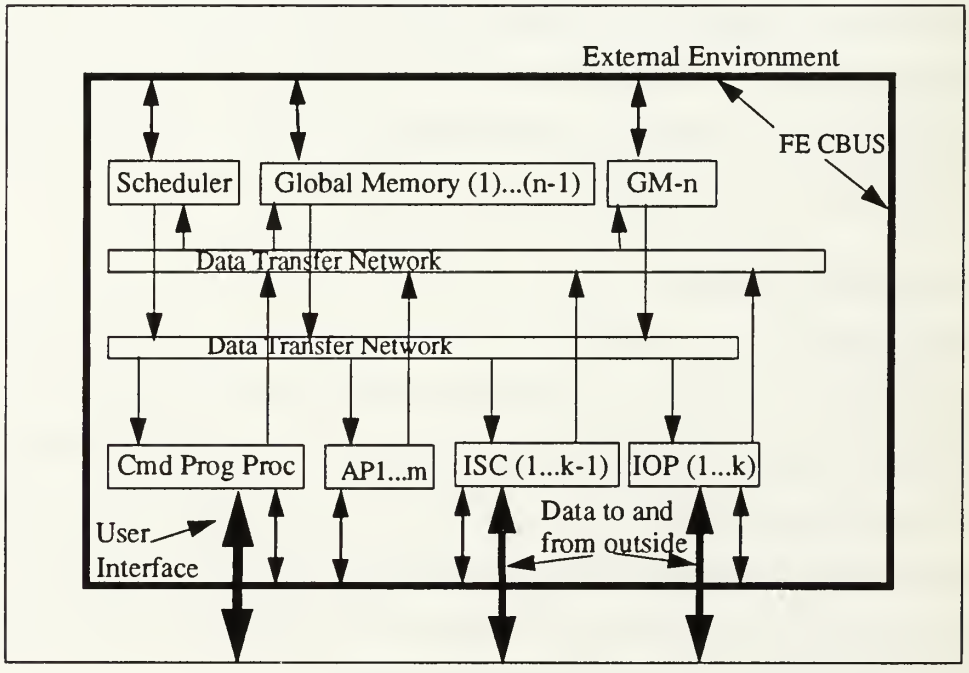


Figure 1: AN/UYS-2 Architecture [Little 91].

1. System Architecture

The system's modular design is based on six different functional elements. These are the Scheduler (SCH), the Arithmetic Processor (AP), the Global Memories (GM), the Input/Output Processor (IOP), the Command Program Processor (CPP), and the Input Signal Conditioner (ISC). Each of these performs specific functions in the architecture and they are all connected by two buses, the Control Bus (CBUS) and the Data Transfer Network (DTN) (Figure 1).

2. The System's Use of Data-Flow

Because DSP algorithms involve minimal decision making they are ideally suited for a data-flow machine. We avoid the inherent penalties involved in multiple branching and can minimize communication overheads if we choose granularity correctly. The data-flow paradigm and its implementation in the AN/UYS-2 are reviewed in the following chapter and a detailed description of the machine is in [Little 91] and [Bell 92].

By mapping nodes of the LGDF graph to processors as their data becomes available we naturally schedule and then execute the algorithm. Nodes are mapped to Arithmetic Processors and edges correspond to the data flows on the DTN and the FE CBUS. Currently the system uses a First Come First Served (FCFS) algorithm to schedule nodes on processors. This approach takes advantage of the inherent strengths of multiple processing by attempting to schedule nodes to any available processor.

3. The Revolving Cylinder Technique

In real-time DSP the two most desired properties are predictability and throughput performance [Little 91]. Unfortunately, the inherent non-determinism of the data flows in a LGDF graph can be exacerbated by an arbitrary policy of resource conflict resolution and thus degrade the predictability of output.

The research efforts of Zaky and Shukla [Shukla 92] of the Naval Postgraduate School seek to improve the efficiency of resource allocation in the AN/UYS-2 and thus effect a reduction in the unpredictability of the DSP's output arrival. The resulting

scheduling technique is called the "Revolving Cylinder". The key idea of the technique is that it inserts synchronization arcs in the LGDF graph in order to improve throughput. It restructures the graph by performing a compile-time analysis of each application execution profile. Each node in the graph is scheduled to run at its earliest possible start time. If that is not possible due to dependencies then it is delayed until the dependency is satisfied. The restructured graph is then mapped to a specific number of AP's to determine whether it satisfies the required data rate. This technique ensures maximum processor usage by only giving resources to those nodes capable of executing at that time [Little 91].

C. OBJECTIVES AND ORGANIZATION

1. Objectives

The purpose of this thesis is to review current research in the field of scheduling real-time applications on data flow architectures and then attempt to find possible improvements to the Revolving Cylinder. The thesis distills the salient features of the Revolving Cylinder technique and establishes a framework of comparison. This becomes a benchmark against which to compare the methodologies of other real-time scheduling research. Current techniques are reviewed and then compared to the Revolving Cylinder with emphasis on the differences, strengths, and weaknesses of each when viewed in the context of the framework.

2. Organization

Chapter II consists of a brief review of Digital Signal Processing and the data-flow paradigm. This familiarizes the reader with the task of the AN/UYS-2 and the reasons a data-flow architecture is so uniquely suited to the task. Chapter III covers the Revolving Cylinder in depth and establishes the primary features of the technique in order to establish a reference framework for comparison with other techniques. Chapter IV covers current research efforts in real time multiprocessor scheduling and compares them using the

framework of the RC as a reference. Chapter V is the conclusion in which recommendations are made and in which future research possibilities are covered.

II. BACKGROUND

A. DATA-FLOW IMPLEMENTATION OF DSP

The military has a number of applications which use digital signal processing techniques in their implementations. These include radar and sonar systems, image processing, speech recognition, etc. Each is of vital strategic concern to the nation given our increased requirements for sensors, control, and intelligence information. The Naval Postgraduate School is working on improving DSP performance for systems operating in real-time environments such as the AN/UYS-2.

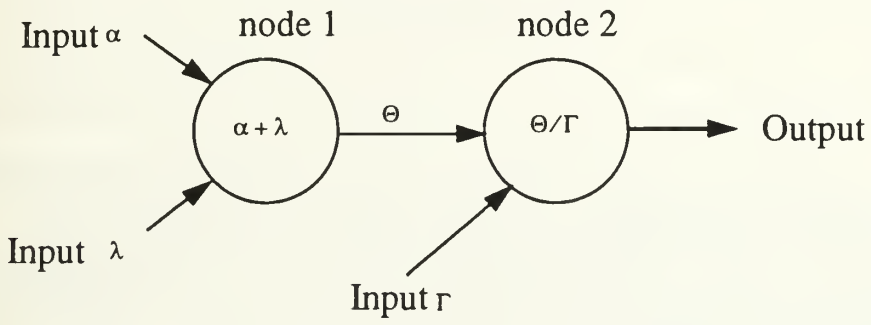
1. DSP Performance

The current and future performance needs of DSP applications require ever-increasing throughput capacities. This necessitates the use of cutting-edge and extremely expensive hardware. Yet as hardware technologies improve we approach the physical limitations of single processor architectures. A processor capable of 1 billion operations per second requires a 1 nanosecond clock period. At this point we start to see the limitations imposed by the speed of light because a signal can only move 20cm in silicon during such a short interval. This causes huge design problems in terms of skewed clock signals, size limitations, and performance degradation [Meng 91].

An attractive alternative to increasing single processor performance is the use of multiple processors concurrently working on a single task. A multiple processor's potential to divide a job up and perform it faster means higher throughput with less expensive hardware.

The first hurdle, however, is that sequential programming languages fail to fully exploit concurrency because the programmer spends a great deal of time countering the basic design of the language by using special instructions designed to spawn parallelism. Development and debugging are difficult because of the contradiction between language structure and programming task. Languages and applications whose properties promote parallelism are thus the easier to implement.

Data flow introduces the notion of values applied to functions rather than instructions fetching the contents of memory cells as in conventional control flow [Gaudiot 87]. Conventional Von Neumann machines declare an instruction ready when a program counter points to it. This event is usually under the direct control of the programmer. A control flow program is a sequential listing of instructions whereas as a data flow program is best represented as a graph in which nodes are instructions which communicate with other nodes using the edges of the graph as illustrated in Figure 2.

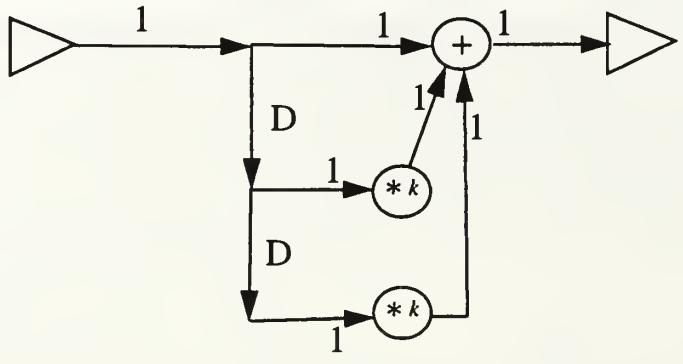


Node 1 must have tokens on both of its input arcs before it can fire. Similarly, node 2 must have the result of node 1 and data coming in on its other arc. This ensures that node 2 waits for node 1 even though they fire asynchronously.

Figure 2: An illustration of a data-flow computation [Gaudiot 87].

Signal processing algorithms are appropriate for description by functional languages and are often represented by mathematical expressions and a graph form (see Figure 3 below). Using Graphical representations of an application allows the programmer to utilize an intuitively obvious representation of a task. A DSP graph is best implemented by a vector operation (i.e., a loop in which all iterations present no dependencies among themselves) which easily delivers parallelism by compiler analysis or programmer

inspection. It usually consists of simple constructs such as arithmetic instructions, FFT butterfly networks, simple filters, and so on.



A Graphical representation of a second order digital filter. The forks replicate each input sample on all output paths. The "D" on two of the arcs indicates delay and the "1"s adjacent to each node indicate that a single token is produced or consumed on that edge when the node fires [Lee 87]

Figure 3: Example of a DSP application's graphical representation [Lee 87].

DSP expressions readily translate into data-flow graphs. An instruction is declared executable when it has all its operands (see Figure 2). We can see the utility of a paradigm which encapsulates nodes so naturally. In the graphical representation above this means that all the input arcs to a node must carry data values (referred to as tokens) before the node is executed. Execution proceeds by first absorbing the input tokens, processing the input values according to the instructions of the of the node, and accordingly producing result tokens on the output arcs [Gaudiot 87]. The graphical representations of a DSP are highly similar to those of a data-flow algorithm and as such map naturally to an architecture using this paradigm.

The graphical description of a digital filter (Figure 3) is a directed, acyclic graph and could be implemented on a data-flow machine. The nodes represent large grain computations which can be selected from a library of signal processing functions.

2. Data Flow Implementation of DSP

Practical implementations of a data-flow approach require some mechanism for both the management of data flows and the capture of the built-in scheduling and synchronization properties of the graph. These mechanisms typically operate at run-time and result in overheads that lead to sub-optimal performance. The amount of overhead depends upon the granularity of the graph and on the amount of recursion or branching present. Research, in fact, shows that a hardware implementation of the data-flow paradigm for general applications results in unmanageable overheads [Shukla 92].

Our problem lies in finding tasks that can use current multiprocessor technology to increase throughput speeds. DSP naturally yields a great deal of useful parallelism because we know, a priori, the amount of data produced and consumed during execution and that there is negligible use of decision making or branching at in the application.

Data-flow graphs describe the dependencies between the different functional nodes of an application. They also provide intrinsic scheduling and synchronization because the executability of an instruction is decided by local criterion only and the presence of the operands is sensed locally by each instruction. This is an attractive property for an implementation running in a distributed environment.

If we choose the granularity of the nodes correctly then the effect of each operation is limited to the production of results consumed by a specific number of other nodes. This precludes the existence of side-effects which may effect the state of a cell of memory used only much later by some other unrelated operation. Granularity has the added benefit of keeping interprocessor communications to a minimum. The generality of this

representation allows us to specify parallelism from the instruction level all the way up to the task level.

B. Data-flow implementation of DSP on The AN/UYS-2

Applications are specified as data-flow graphs with nodes representing large grain computations chosen from a library of signal processing functions. The edges of a graph represent queues which receive data from the source node and supply data to the destination node. Each queue is allocated a memory module for storage which maintains its current size and remaining capacity.

As data arrives on all the input queues of a node, the threshold values (the minimum number of data items that must be present in a queue for its destination to become ready) associated with each queue are eventually exceeded. A node is ready for execution when two conditions are satisfied:

(1) All incoming queues exceed their thresholds and

(2) all output queues must be under their capacity values.

All memory modules communicate the events of threshold/capacity crossing to the scheduler which determines if a node is ready. Initially all processors are on the Free Processor List (FPL) and the scheduler assigns them nodes as they are placed on the Ready Node List (RNL).

1. Setup, Execution, and Breakdown

When a node is assigned to a processor it fetches the data and the instruction stream corresponding to the node from the appropriate memory module. When the entire instruction stream and queue data are fetched the setup of the node is complete. Each processor communicates this event to the scheduler to get itself placed on the FPL so that the next node may start setup. Thus, the node already setup begins execution while the next node on the RNL begins setup. This occurs under the restriction that a processor may have only one node set up and pending to execute at any time. The data generated by an

execution is first stored locally. Upon completion, a processor transfers the data to the appropriate memory-module storing the output queues in what is referred to as the breakdown phase.

Every node goes through three phases at a processor: Setup, Execution, and Breakdown. Since their functions are independent and the set-up/breakdown operations may require time comparable to the execution time, these operations could be overlapped by providing independent functional units for data movement and execution in the processor.

2. Performance Degradation

Upon arrival of sufficient data at nodes which only receive input from the outside world, an instance of the graph is started and its execution proceeds according to the data-flow principle. As a result of the data-flow execution, which corresponds to asynchronous task-level pipelining, several instances of the graph are active simultaneously.

Aside from the requirement that the required throughput must be met by the machine, real-time performance may require that all instances of the graph should complete in the same amount of time. Between the completion of the setup of a node at a processor and the actual start of its execution, there may be a delay because the execution unit at a processor has not completed the previous node. This delay is in addition to the delay a ready node may experience waiting on the RNL. Both delays result in an increase in the latency of the graph execution.

On the other hand. an execution unit may have to wait for the setup completion of the next node assigned to it after it completes its current node. If this happens, execution cycles are lost and the machine's throughput degrades.

To maximize throughput all execution units must run continuously so each processor must have a node set up for execution at the time it finishes the previous node's computation. Because the scheduler is a simple run-time dispatcher that matches RNL

nodes to free processors, the delays described above depend upon the application's execution profile. This profile depends upon the data rate, the spatial and temporal parallelism in the graph, the number of processors in the system, the number of memory modules, and the allocation of queues to memory modules.

Since task-level parallelism is being considered, performance can be improved significantly if setup and breakdown cost can be minimized. One method to reduce this cost is to chain successive nodes together and execute them on a single processor one after the other. This results in saving the breakdown cost for the first node and setup cost for the next node.

C. Unpredictability in Program Behavior

In real-time environments the ability to predict a program's performance is critical for efficient allocation of resources such as memory modules, processors, and queue sizes. The AN/UYS-2's use of the First-Come-First-Served (FCFS) paradigm for assignment of processors to ready nodes degrades its performance in two ways: Irregular execution patterns and interference/contention in the memory modules.

When data arrives periodically, unpredictable execution patterns arise due to the absence of direct control over execution of nodes that depend only upon the receipt of data from the external world. If the output queue capacities for these nodes are unlimited they execute at a rate that matches the input arrival rate and are independent of the rate at which other nodes execute. In the presence of finite queues, they execute at the input rate until the output queues are filled and then stall until nodes down the graph create space in the queues by consuming data from the output queues. This leads to the individual graph instances not being executed uniformly. This is undesirable in real-time environments because it leads to non-deterministic output rates and thus cannot guarantee that minimum performance bounds will remain inviolate.

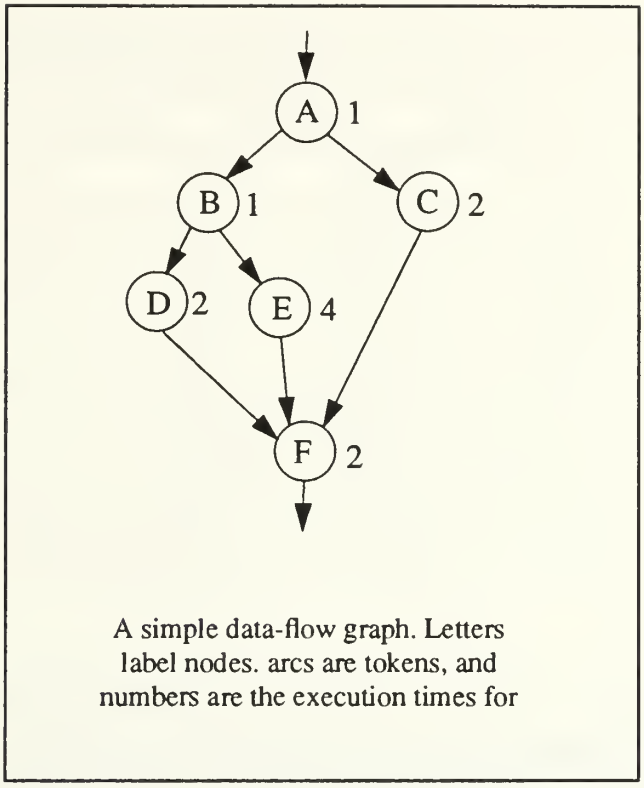


Figure 4: A sample input graph for the AN/UYS-2 [Little 91].

Figure 4 is an illustration of a simple data-flow graph and Table 1 is a possible schedule of execution for that graph. The table shows how the schedule might run in an environment in which the inputs from the outside world readied an "A" node for the RNL on every cycle. Without any additional scheduling management the RNL swiftly fills with the second and third instances of the graph before the system has a chance to fully execute the first instance. FCFS guarantees that the first instance of a graph will finish before the next but it cannot provide anything close to deterministic output as it approaches heavy loads Machine throughput can degrade because the memory access patterns may be such that there is contention at the memory modules while setting up and breaking down nodes.

TABLE 1: A possible execution of the graph of Figure 4 under FCFS

Cycle	AP 1	AP 2
1	A1	
2	B1	C1
3	A2	C1
4	D1	E1
5	D1	E1
6	D1	E1
7	A3	E1
8	B2	C2

In the following section, a framework is presented that introduces synchronization dependencies in the graph based on the technique of revolving cylinder analysis. This technique addresses the problem illustrated above by inserting extra dependencies in the graph and then enforcing them at run-time. In this way we avoid much of the overhead of run-time scheduling management by using the execution profile of the graph to do the work for us.

III. Revolving Cylinder Analysis

Revolving Cylinder Analysis generates a new data-flow graph as a result of compile-time analysis [Shukla 92]. This provides built-in run-time support for the system scheduler. The Revolving Cylinder restructures the application, described as a task-level data-flow graph, by mapping it on the surface of a hypothetical cylinder whose dimensions are determined by both the number of system processors and the sum of node execution times.

The technique results in increased predictability in simulations of typical DSP applications [Shukla 91]. It differs from other research in that it uses the application profile of the graph to reduce the scheduling overheads that make data-flow so difficult to implement. The essential features of RC analysis are outlined in this chapter in order to establish a framework of comparison with current research in this field.

A. An Introduction to RC Analysis

The key to RC analysis is that the insertion of dependencies in the application graph will result in both increased throughput performance and more deterministic output rates. These added dependencies change the point at which a node will enter the Ready Node List (RNL) based on whether or not its predecessors higher in the LGDF graph are complete and whether previous iterations of the graph are complete. The actual scheduling of a node to a processor is left to the scheduler (SCH) at run-time. The goal is to allow scheduling to remain dynamic and thus keep overheads low.

The Revolving Cylinder automatically determines whether an application can meet real time requirements during graph compilation. Having done so it then restructures the graph so that it will have more deterministic throughput and output arrival rates. This ensures that each instance of a node completes without the creation of an execution backlog in the lower nodes as discussed in Chapter II.

Given the simple application graph in Figure 5, RC analysis determines whether it can be mapped to a set number of processors while still satisfying a required data rate. For reasons of brevity the costs of setup and breakdown for each node are ignored.

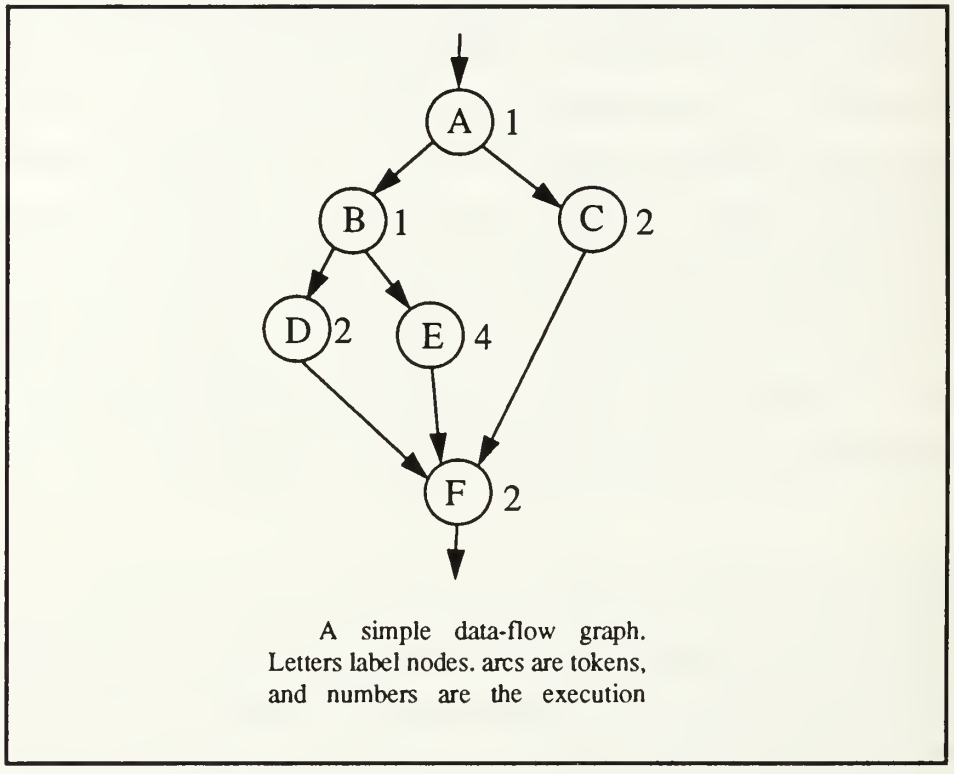


Figure 5: Reference data-flow graph [Little 91]

It can be proved that, as long as communications overheads are ignored, the optimum throughput for an application is the sum of node execution times divided by the number of available processors. As an example, a system with 2 processors executing the graph of figure 5 has an execution time of (12/2 = 6) cycles. The optimum result is that the system could start a new instance of the graph every 6 cycles as long as it avoids the scheduling pitfalls akin to those of FCFS discussed in the previous chapter [Little91].

B. Insertion of Delays

The idea of delays in the execution graph provides a stepping stone to the concepts of the revolving cylinder. If we insert artificial delays into the graph we can overlap the execution of subsequent instances of a node because the delays force the graph to execute uniformly despite the fact that some nodes may have their data available before others. Using the simple application graph of the previous example as a starting point we insert the delays required to ensure that an instance of the graph can be executed and overlapped every six cycles. The altered graph is shown in figure 6.

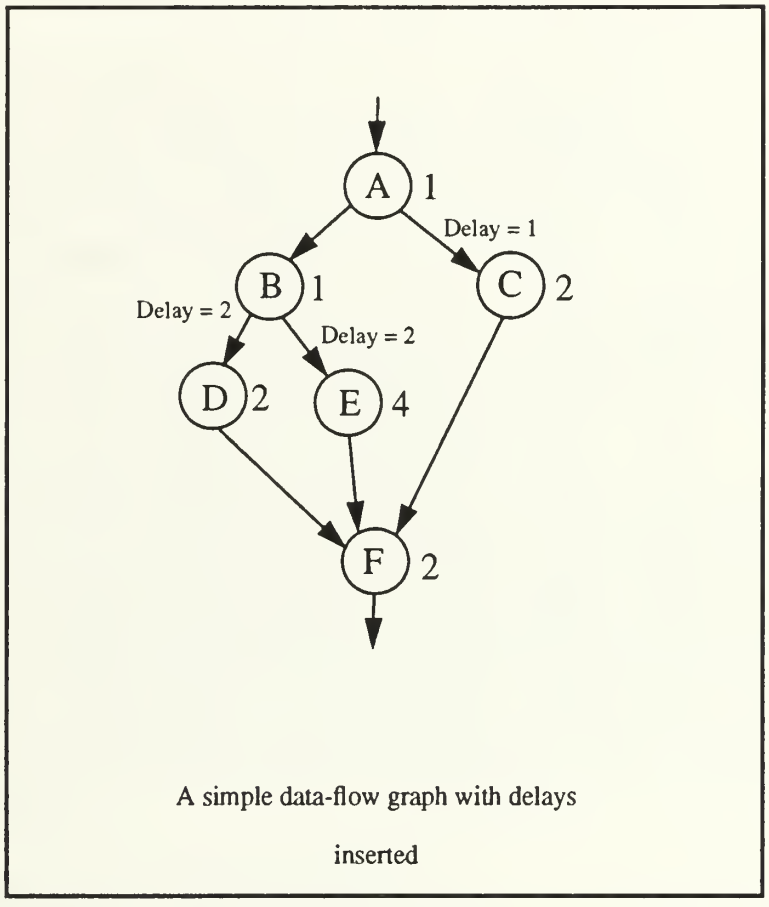


Figure 6: The graph of Figure 5 with additional delays [Little 91].

The delays seem counterintuitive for improved performance until we realize that they facilitate the control and execution of multiple instances of an application [Shukla92]. They help control the execution of the graph by forcing the system to wait on execution of a node until the nodes higher in the graph are begun. Table 2 depicts the schedule table of one instance of the application of Figure 5, with delays, executing on the system. By inspection of the schedule we see that another instance can be started every six cycles because the delays keep the execution of the graph free of the latencies found in the FCFS algorithm.

TABLE 2: A template for the execution of the graph of Figure 6

Cycle	AP 1	AP 2
1	A1	
2	B1	
3	C1	
4	C1	
5	D1	E1
6	D1	E1
7		E1
8		E1
9		F1
10		F1

TABLE 3: Execution profile of the RC schedule for Figure 6 at any point after start-up

Cycle	AP 1	AP 2
6(i) - 5	A(i)	E(i - 1)
6(i) - 4	B(i)	E(i - 1)
6(i) - 3	C(i)	F(i - 1)
6(i) - 2	C(i)	F(i - 1)
6(i) - 1	D(i)	E(i - 1)
6(i)	D(i)	E(i - 1)

With the exception of the first 6 cycles of the schedule, which represent a transient, every subsequent group of six consecutive cycles could be summarized by the schedule in Table 3. With this paradigm we are almost at the heart of the Revolving Cylinder but for one important difference. The artificial insertion of delays works well as a run-time scheduling mechanism but it is difficult to implement during compile-time analysis. We want a simple technique which will take advantage of the inherent scheduling of the graph at compile-time so as to keep run-time overheads low.

C. Implementation of The Revolving Cylinder

RC scheduling recommends when a graph node is scheduled at compile-time (i.e.: statically) but choosing the AP to schedule it on is left to the run-time dispatcher. This enables execution scheduling to remain dynamic. The reason for implementing the algorithm as a cylinder is that data arrives periodically and so the application is invoked cyclically [Little 92].

1. Mapping Nodes to The Cylinder

The idea is to schedule the graph such that it wraps around the cylinder and its end meets its beginning. Let us assume that there is a cylinder whose circumference is the intended execution length of the schedule in Table 3 (6 nodes with a total of 12 cycles to

be executed in the example) and whose height is the number of processors (2). The table can be wrapped around the cylinder such that its beginning meets its end. The line on the surface of the cylinder that separates the end from the beginning has the effect of a divide-by-C counter where C is the circumference of the cylinder. The counter is incremented every time the line is crossed to enter the beginning from the end. Hence we get the counter, (i), which allows us to keep track of which nodes belong to any particular graph in execution. [Little91].

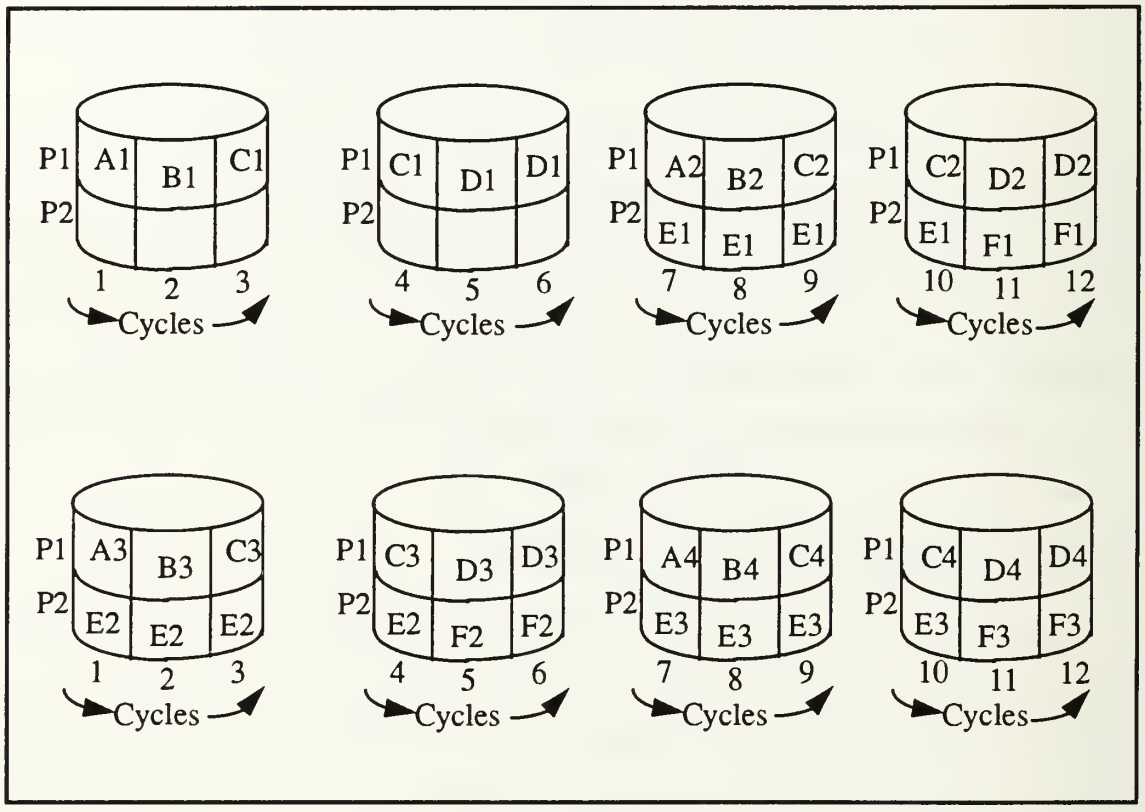


Figure 7: A visualization of the graph of Figure 6 executing on a Revolving Cylinder [Akin 93].

Figure 7 is an illustration of the schedule of Table 3 mapped to a Revolving Cylinder. The transient start-up cost of the schedule is prohibitive and seems disproportionate were the application executed only once or twice. The benefit comes once

the machine gets past the 7th cycle. The run-time enforcement of this mapping ameliorates the nondeterministic output rates of data-flow graphs. It is readily apparent that, although each instance still takes 12 cycles, the system will complete an application every six cycles and thus reach the full potential offered by two processors. In this the Cylinder operates much like asynchronous pipelining on a control-flow machine [Akin 93].

Each slot in the cylinder is of width equal to the smallest node in the graph. For each node in the graph, starting with the top node (in our example, A) and working towards the bottom node (F), attempt to schedule the node at its earliest start time. If it cannot be inserted at start time, delay the start time by the width of a slot and repeat until it can be inserted. Adjust the earliest start time of all descendants of that node and repeat the sequence with the next node as the top node in the graph. In the same way that delays helped in the previous section this mapping ensures that maximum cylinder usage (and hence throughput) will result.

2. Assigning Scheduling Arcs in The Graph

Once all nodes have been inserted into the cylinder and the cylinder is full, assign arcs to the nodes based upon their location in the cylinder. For each entry mapped to an AP in the cylinder, if there are other nodes assigned to the same AP with the same index and the node higher up in the cylinder is not an ancestor of the other, then create a dependency from the higher node to the lower. The restructuring of the graph in the example is not unique. There are several ways of filling the table and so there are corresponding sets of additional dependency arcs.

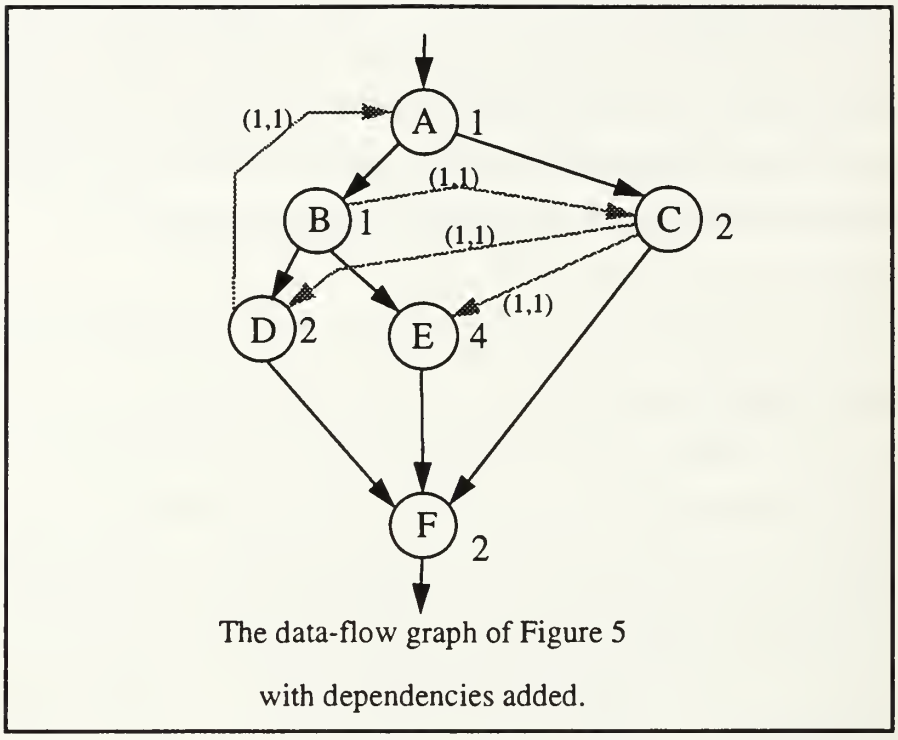


Figure 8: Graph of Figure 5 with added scheduling arcs [Little 91].

Even for a single assignment, there exist several sets of additional dependencies. This introduces the problem of selecting the best assignment and a suitable set of arcs associated with it for some arbitrary graph. The heuristic used for such selection is minimization of the number of additional arcs introduced. Figure 8 shows one possible restructuring resulting from this technique.

The run-time mechanism of the scheduler is fixed and thus any execution sequence enforcement is accomplished at compile-time. The grey lines in Figure 8 show the additional data-dependencies used to enforce RC assignment at run-time. Each grey line represents a queue of tokens generated by the source and absorbed by the destination. Each source generates a single token when it completes execution. The 2-tuple associated with each indicates the threshold and consume amounts for the control token flow on these arcs. The threshold amount refers to the number of tokens that must be present on the arc for its destination node to be eligible for execution.The consume amount refers to the number of

tokens removed from the arc when it executes once. Thus, the arc from B to C forces node C to delay going onto the Node Ready List until node B is complete. This has the same effect as specifying delays without actually scheduling them into the application graph.

Given such restructuring, the setup and breakdown times for arcs (A,B), (B,D) (A,C) and (E,F) can be minimized. This is done by chaining sequential nodes which feed directly into each other. The nodes are collapsed into a single node for assignment to a single processor. The trade-off becomes one of reduced overheads for communication versus loss of parallelism and throughput gains. The flexibility of the system's granularity enables the system to make this choice effectively. It is assumed that the overhead of implementing the control-token queues is negligible with respect to the cost of implementing data queues [Levine 92].

D. Framework for Comparison

Based on whether a scheduling decision is made at compile-time or at run-time we can classify a data-flow implementation over a spectrum that ranges from fully static to fully dynamic. Dynamic implementations have the most management and communication overhead but this makes them more flexible and easier to implement than a static implementation. They have the added benefit of being more robust in the case where a processor malfunctions and so degrade gracefully.

To their credit, static implementations are more efficient and have the predictable performance crucial to a real-time system. They are, however, difficult to realize, inflexible, and degrade poorly. Their effectiveness is determined by how accurately the computational problem is known before-hand. This is a difficult problem and typically the worst-case estimate results in large inefficiencies.

A carefully implemented hybrid of compile-time effort and run-time complexity strikes the appropriate balance between throughput and guaranteed performance. RC analysis provides such a blend by building scheduling management into the graph at

compile-time and then allowing the run-time scheduler to assign nodes to processors dynamically.

A node is synchronous if we know, a priori, how many new input samples are consumed and how many output samples are produced every time a node is invoked. A Synchronous Data-Flow graph is a directed acyclic graph made up of synchronous nodes [Lee 87]. Revolving Cylinder analysis is most suited for use with synchronous data-flow graphs.

The RC technique is directed towards improving throughput and the determinism of output flow in real-time systems, under high loads, with repetitive tasks to perform. Tasks that fall in this category are those such as radar, Magnetic Resonance Imaging, and other continuous scan applications. Other real-time scheduling systems are concerned with getting the fastest possible response without regard to how efficient the continued execution of the task might be. These fall under the guise of some weapon systems applications in which instant response is required from a single instance of an application. These schedulers seek to pack an application graph so that it will run in the least possible number of cycles.

The system we use is non-preemptive. Enough research is available in the literature to obviate an extended discussion of this thesis. Suffice it to say that the graph's inherent structure implies nodal orders of execution. This, combined with known node execution times, leads to more deterministic output flow than a preemptive scheduling scheme. Figure 9 illustrates the difference between the two [Lewis p.249]:

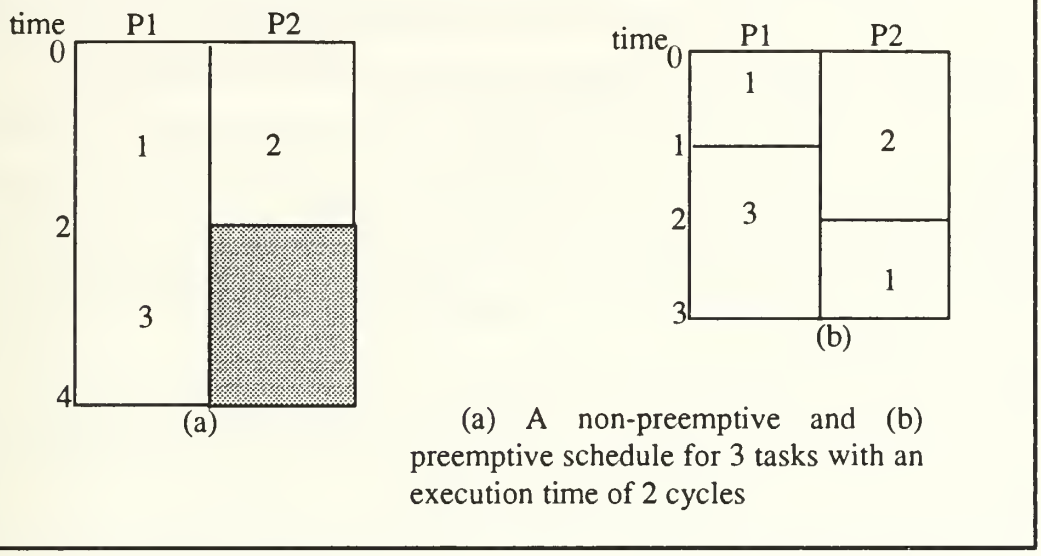


Figure 9: Preemptive vs. non-preemptive scheduling.

Revolving cylinder analysis is a policy which can be implemented on a number of different machines. The key is that it improves the determinism of output flows whenever there are repetitive tasks whose executions are deterministic. It does this by a mix of static scheduling and dynamic assignment of nodes to processors at run time. We are interested in the approaches used by other researchers in the field of real-time scheduling. Chapter IV covers these in detail.

IV. ALTERNATE APPROACHES

We now look at data-flow graph scheduling techniques ranging from the scheduling approach used to implement real-time prototypes on the Naval Postgraduate School's Computer Aided Prototyping System (CAPS) to Som's multiprocessor "Algorithm To Architecture Mapping Model". Each of these seeks to improve real-time performance of systems using directed acyclic graphs. The target architectures vary from simple control flow von Neumann machines to a SSIMD architecture. This chapter covers the approaches in depth and discusses the strengths and weaknesses of each.

A. Scheduling Hard Real-Time Systems on CAPS

1. An Introduction to CAPS

The Computer Aided Prototyping System (CAPS) being developed at the Naval Postgraduate School seeks to overcome the complexity in the design and development of hard real-time environments using rapid prototyping to build and maintain these systems [Levine 91]. Rapid prototyping is a means for stabilizing and validating the requirements for complex systems (e.g. embedded control systems with hard real-time constraints) by helping the customer visualize system behavior prior to detailed implementation. CAPS supports an iterative prototyping process characterized by exploratory design and extensive prototype evolution, thus enabling engineers to produce complex systems that match user needs and reduce the need for expensive modifications after delivery [Levine 92].

2. System Overview

CAPS consists of several modules. Figure 10 describes the major software modules of the system.The first module of the system is the user interface which consists of a graphical editor for the formal prototyping language called Prototyping System Description Language (PSDL). The second module is the Software Database System which

includes the Rewrite Subsystems, the Software Design Management Subsystem, and the Reusable Software Component Database.

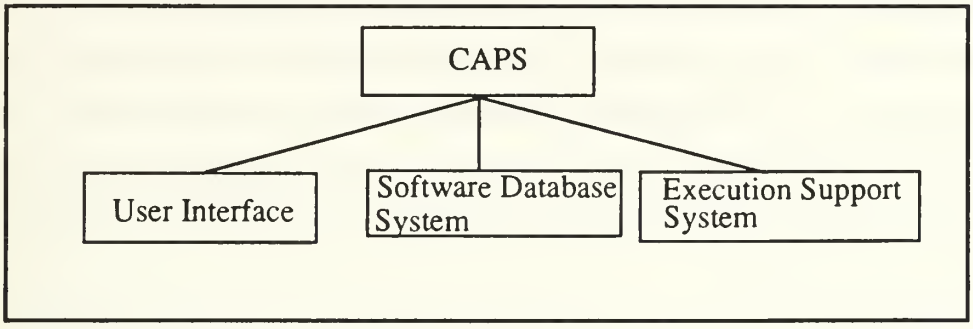


Figure 10: CAPS modules [Levine 91].

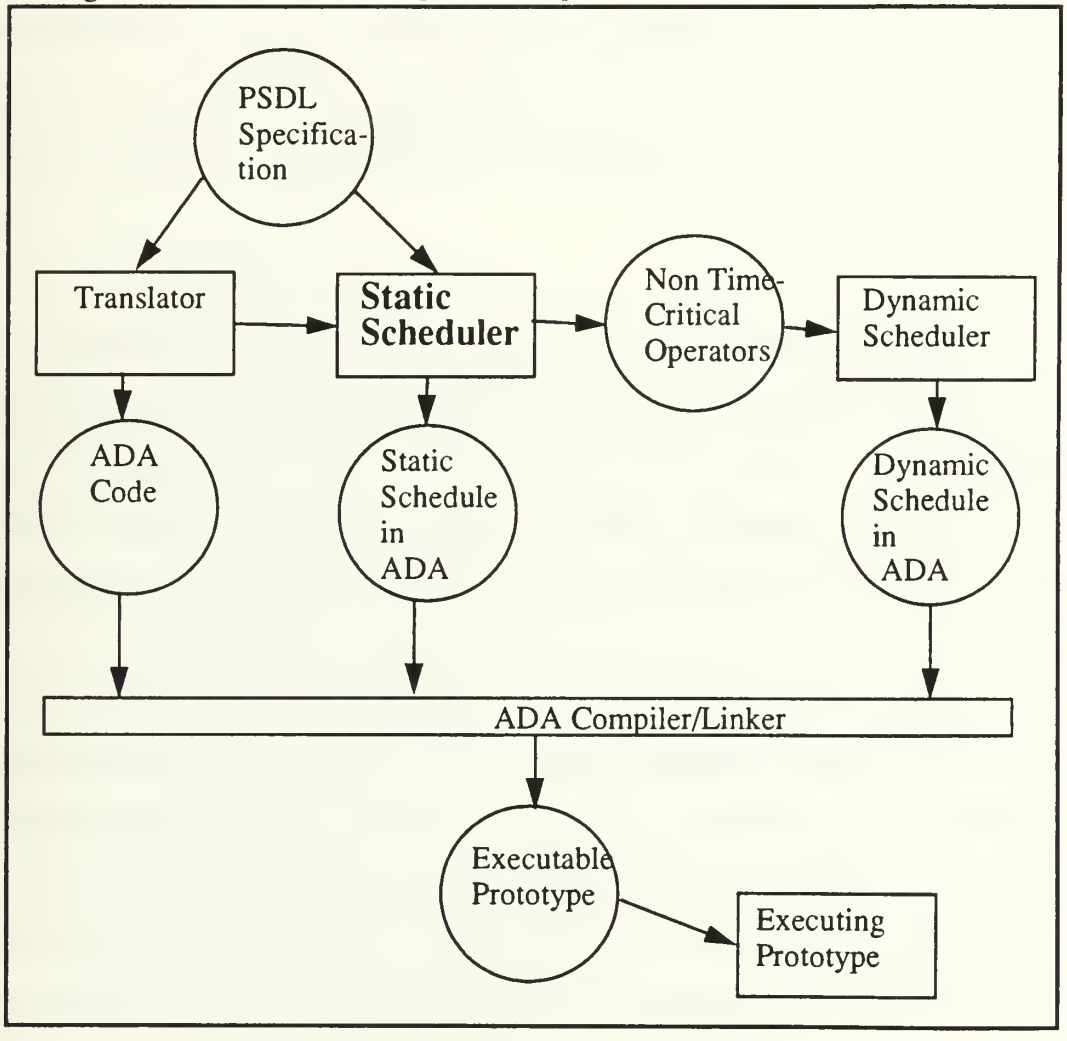


Figure 11: The Execution Support System (ESS) [Levine 91].

The third module is the Execution Support System (ESS). This module contains the PSDL Translator, the Static Scheduler, and the Dynamic Scheduler. Figure 11 shows the implementation and interfaces of the ESS.The Dynamic Scheduler acts as a run-time executive when exercising the system. It schedules nodes without timing constraints, which are not included in the static schedule, by using spare capacity or slack in the static schedule (see Figure 14). It handles run-time exceptions and hardware/operator interrupts and communicates with the user interface during prototype runs. Thus it performs like a miniature operating system.

It is the static scheduler that we are interested in. The purpose of the static scheduler is to build a static schedule for a set of tasks that must obey both precedence and timing constraints. This schedule gives the order of execution and the timing of the operators. It is legal and feasible if both precedence relationships are maintained and timing constraints are guaranteed to be met.

3. The Static Scheduler

The static scheduler has five modules: PSDL READER, FILE PROCESSOR, TOPOLOGICAL SORTER, HARMONIC BLOCK BUILDER, and OPERATOR SCHEDULER.

The first component, PSDL READER, reads and processes the PSDL prototyping program. It is essentially a filter that removes information not needed by the static scheduler.

The second, FILE PROCESSOR, analyzes the text file generated by reader and separates the information into a linked list data structure and a file of non-critical nodes. It then converts sporadic operators into their periodic equivalents. The block builder and the operator scheduler generate linked lists containing the vertices and links of the graph.

The third component, TOPOLOGICAL SORTER, performs a topological sort on the data structure. It develops a true topological ordering and is not dependent on a

specific ordering of nodes in the PSDL input file. The result is a total ordering of the nodes depending on data flow.

The fourth component, HARMONIC BLOCK BUILDER, determines the Harmonic Block length of the static schedule. An illustration of the Harmonic Block is found in Figure 14. The system takes each of its real time processes and finds their least common multiple. This guarantees that the system will schedule and execute each critical process within the bounds of performance. The trick is to find a harmonic block which will meet the performance constraints of a real-time system.

The last module, OPERATOR_SCHEDULER, combines the output of TOPOLOGICAL_SORTER, FILE PROCESSOR, and HARMONIC BLOCK BUILDER to produce a static schedule. The static schedule is a linear table giving the exact execution start time for each time-critical node and the reserved maximum execution time (MET) for each.

4. Graph Implementation

The nodes are atomic or composite. An atomic node is defined as the basic individual unit of work to be executed and a composite node is defined as being a node that can be decomposed into atomic nodes. This allows the system to deal with varying granularity. Each node is characterized by its timing constraints, precedence constraints, and resource constraints. The researchers assume that the resource requirements for each node, to include memory and external systems, are always met.

There are two different types of data in PSDL: discrete and continuously sampled streams. Discrete data are used in applications where the values of data must not be lost/replicated and in which the period of the producer and consumer of the data must be the same (lockstep performance). Sampled data are used in applications where values must be available at all times and can be replicated without affecting their meaning. Each data stream represents a directed edge from the node that produces the data to the node that

consumes the data in the precedence graph. Cycles, and hence internode recursions, are not permitted in the precedence graph.

5. Creating the Schedule

a. Algorithm Options

After creating a constraint graph the static scheduler creates a schedule using one of the following algorithms: Earliest Start Time, Exhaustive Enumeration, or Simulated Annealing. Since the static scheduling problem is NP-hard, systemic global search is the only guaranteed way to return a feasible static schedule for a hard real-time system if such a schedule exists. The exhaustive enumeration algorithm is implemented in CAPS to accomplish this, but the algorithm is very costly in practice.

Shing and Levine [Levine 92] developed a simulated annealing approach as a heuristic algorithm to schedule the prototypes of hard real-time systems. The goal of this algorithm is to quickly find a valid schedule if one exists in a majority of cases where the cost of complete enumeration is too great.

b. Simulated Annealing

The simulated annealing procedure was chosen because it was iterative, probabilistic, simple and insensitive to the form of the cost function. An example combinatorial optimization problem is an assignment problem where there are a number of personnel available to do an equal number of jobs. The cost for each person to do each job is known. The goal is to assign each person to a job so that the total cost is as small as possible. There are a wide range of combinatorial optimization problems in a similar vein for which simulated annealing is tractable. These include graph partitioning, graph coloring, number partitioning, VLSI design, and travelling salesman type problems [Levine 92].

6. Basis of The Algorithm

Simulated annealing is based on the behavior of physical systems. The approach is modelled on the way that liquids freeze and metals crystallize. At high temperature, molecules move freely with respect to one another. As the liquid cools, this mobility is lost. Atoms line up and form a pure crystal that is at a minimum energy level. As the system cools it tends toward a state of minimum potential energy.

7. Annealing and Optimization

Examining simulated annealing in non-physical terms, a comparison is made to the concept of local optimization or iterative improvement. Local optimization repeatedly improves an initial solution until no further improvement of the solution is possible. This is known as iterative improvement or "hill climbing." Simulated annealing differs from local optimization in that random uphill movements (acceptance of a worse solution) are permitted while the system "temperature" is warm enough to allow it.

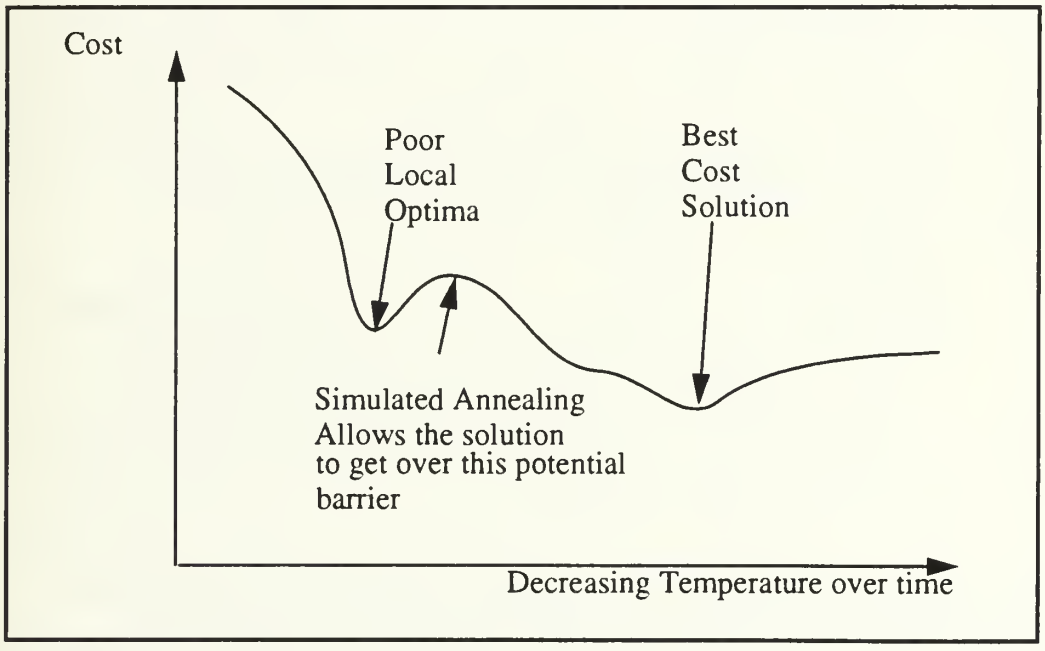


Figure 12: A representation of a simulated annealing solution's cost over increasing time [Levine 91].

This prevents the algorithm from being trapped in a poor locally optimal solution as demonstrated in Figure 4.3. Simulated annealing provides significantly better results than can be found utilizing local optimization.

The key to the use of the simulated annealing approach to solving combinatorial optimization problems is the random acceptance of worse iterative solutions. Initially when the system is in a high energy state (high temperature), the probability is greater that a worse iterative solution is accepted. As the system cools this probability decreases, but even at the lower energy states the probability for making an uphill move still exists. Uphill moves allow the algorithm to leave a poor local solution and reach a better solution. This general scheme of always taking a downhill step while occasionally taking an uphill step is known as the Metropolis algorithm [Levine 91].

8. The Cost Function

The choice of a probability function to determine if an uphill movement is allowed is an important consideration. At each step of the simulated annealing algorithm a new state is constructed based on the current state. This new state is constructed by displacing or adjusting a randomly selected element. If this new state has a lower cost than the current state, the new state is accepted as the current state. If the new state has a higher cost than the current state, the new state is accepted with the probability:

$\exp(-\Delta e/kT)$

This function is known as the Boltzman probability distribution where:

Δe = difference in cost between new state and current state

k = Boltzman's constant of nature relating temperature to energy

T = Current Temperature

A characteristic of this probability function is that at very high temperatures every new state has an almost even chance of being accepted as the current state. At low temperatures the states with a lower cost have a higher probability of being

accepted as the current state. Simulated annealing is simple to implement and can be applied to a variety of combinatorial optimization problems.

9. Real-Time Scheduling Constraints

Developing hard real-time schedules using simulated annealing requires that several modifications must be made to the steps of the simulated annealing algorithm. These changes are required because true random orderings of graph nodes cannot be maintained since there are precedence constraints in a hard real-time schedule. Another change to the algorithm is that hard real-time scheduling only seeks a feasible schedule, not the best possible or optimum schedule. This factor simplifies and speeds up the development of the annealed schedule.

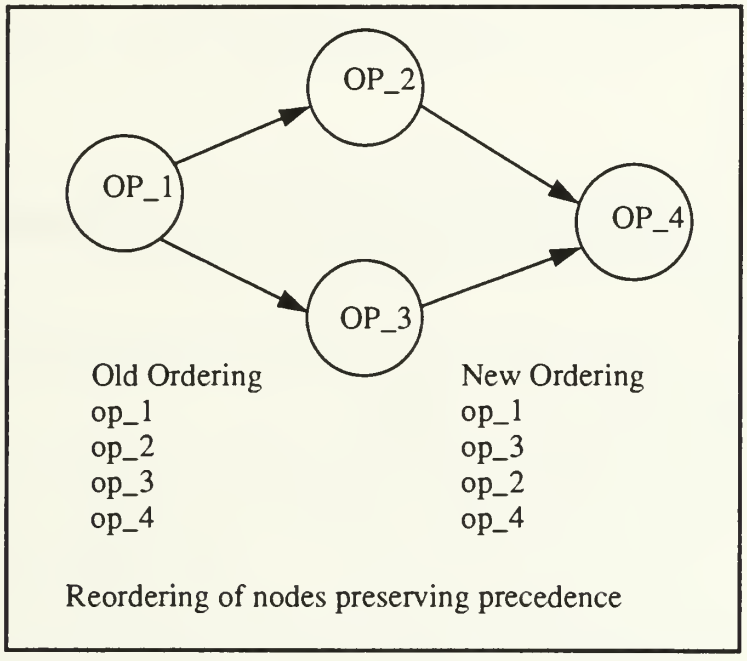


Figure 13: Reordering of nodes using CAPS scheduler [Levine 91].

The method of adjusting a given solution maintains the precedence relationships that exist between operators of a hard real-time system's application graph. As long as precedence is maintained nodes can be adjusted randomly within a given schedule. True random orderings cannot occur since a parent must always appear before its children.

Figure 13 demonstrates a feasible reordering of nodes that can occur using simulated annealing.

In both the old and the new ordering, the position of each operator in the list is valid based on the precedence relationship indicated by the graph. The algorithm guarantees precedence by checking for a parent-child relationship between nodes it is attempting to reschedule. The goal of the hard real-time scheduler is to find a feasible schedule for the graph, not the optimum schedule. This means that the search for a schedule is terminated as soon as a feasible schedule is found. Both loops of the annealing algorithm are modified so that if a feasible schedule is found, the loop condition for both loops is satisfied and annealing is terminated.

10. Solution Deadlines

Each proposed solution, including the initial solution, is examined to see if it satisfies two criteria:

(1) Examine each node's start time.

The start time must be examined to see if any node starts before its earliest allowable start time.

(2) The finish time is then examined to see if it exceeds the upper bound for node termination.

If the upper bound for a node is violated, the amount of time that this bound is violated will be added into the schedule's cost.

a. Precedence

There is no requirement to examine a schedule to see that precedence is maintained since each adjustment to the schedule will guarantee that operator precedence is maintained.

b. Harmonic Block Length

The proposed schedule must also be examined to check that the finish time of the last operator in the schedule does not exceed the harmonic block length. A harmonic block is defined as a set of periodic operators, where the periods of all component operators are exact multiples of the base period. The base period is the greatest common divisor of all periods of the critical periodic operators and the harmonic block length is the least common multiple of these operators as in Figure 14. The basic idea is that a schedule is developed to fit inside a harmonic block. Once a schedule is developed that fits within the harmonic block, subsequent copies of the block can be made to maintain the hard real-time schedule [Levine 92].

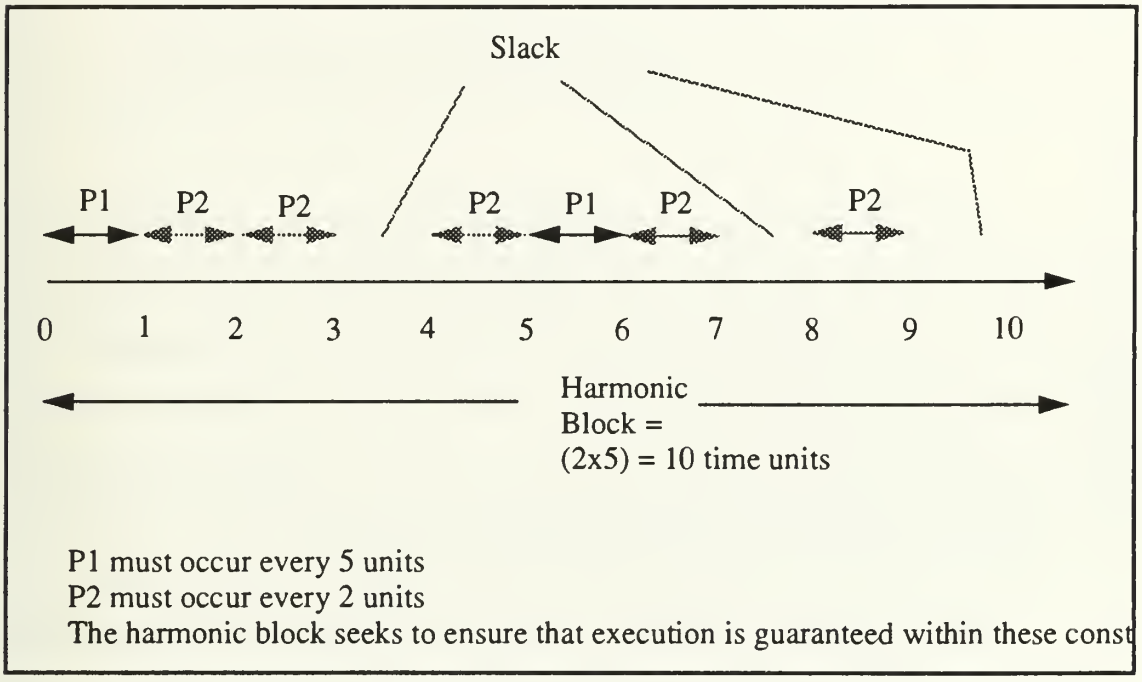


Figure 14: Harmonic Block length in CAPS

If a schedule does exceed the harmonic block length, it cannot be valid because subsequent copies of the schedule will also violate their timing constraints. If the schedule satisfies all timing constraints and the harmonic block length is not violated then

it is feasible. At this point the simulated annealing algorithm is terminated and the schedule is returned to CAPS.

The intent of scheduling real-time systems on CAPS is to guarantee the execution of tasks on a serial processor within a specified time bound. Thus the harmonic block ensures time for each critical process. If a feasible schedule is found (i.e., a harmonic block which satisfies time and execution constraints) the system is going to guarantee that a real-time application will execute within its bounds.

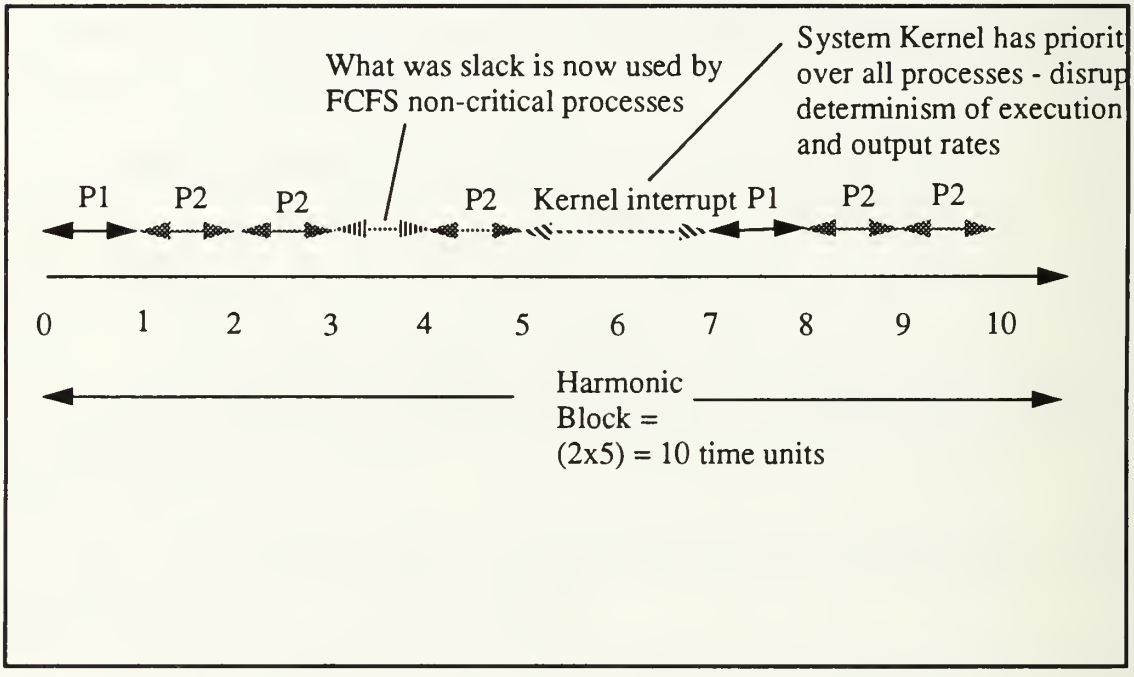


Figure 15: Kernel interrupts and non-critical processes

Real-time processes are given a higher priority than non-critical tasks and so execute within the bounds of the harmonic block. The system handles data arrival both periodically and aperiodically by the use of interrupts and polling. Aperiodic data arrival means that interrupts are necessitated by the arrival of critical tasks with higher priority than an non-critical task currently executing on the CPU. Polling is used to handle the execution of queue of non-critical tasks waiting for slack in the execution of the harmonic block. In periodic operation the system only has to handle the task of polling each of the

non-critical processes competing for system resources. Real-time processes are guaranteed processor time by the Harmonic Block and need no polling.

One of the problems of the system is that the kernel has priority over all tasks as shown in Figure 15. In this example the Harmonic Block of Figure 14 is interrupted by a system call. Once the system call is finished the scheduler crams processes into the block to try and make execution time limits. even if a critical task is in the middle of execution then it is preempted by any kernel calls. A statistical analysis can determine the frequency of these interrupts but there is still non-determinism in the schedule's output flow. Another potential problem lies in the inherently non-deterministic output flow of ADA. There is no way to guarantee performance of the system when no time bounds are guaranteed on the connection interface of ADA sockets, etc. This is a temporary problem being addressed in the next versions of the language but it does bear inspection. More information on the approach is available in [Levine 92].

B. Scheduling for Real-Time DSP Performance on a Rectangular Grid

Lincoln Laboratory of M.I.T. developed a Block Diagram Compiler (BDC) designed and implemented for converting graphic block diagram descriptions of signal processing tasks into source code to be executed on a Multiple Instruction - Multiple Data Stream (MIMD) array computer [Ziss 87]. The compiler takes a block diagram of a real-time DSP application as input entered from a graphics workstation. It then translates the graph representation into code for the target multiprocessor array. The current implementation produces code for a rectangular grid of Texas Instruments TMS32010 signal processors built at Lincoln Laboratory but the concept can be extended to other processors or geometries.

1. Target Hardware Implementation

The current hardware implementation of the MIMD array consists of a two-dimensional rectangular grid of TMS32010-based processing cells. The size and shape of

the array is somewhat arbitrary with the restriction that one cell can be nearest-neighbor to no more than four other cells. Enough communications paths exist in this array to allow it to function as both a 4x4 square grid and a 16x1 linear array (Figure 16).

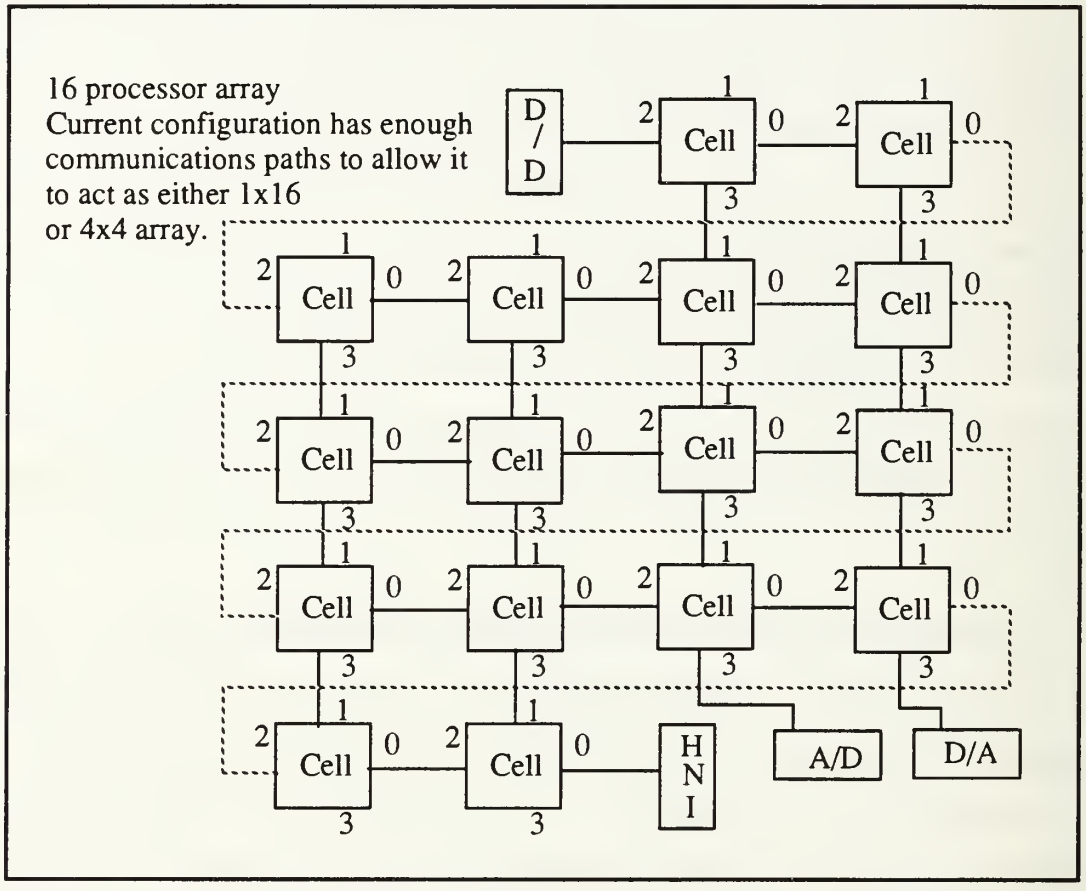


Figure 16: Lincoln's variable geometry MIMD machine [Ziss 87].

2. Mapping The Graph Nodes to Processors

a. Individual Processors

A user begins by drawing a block diagram of his application using a library of basic DSP functions implemented as nodes. The nodes can be as simple as adders and multipliers or as complex as FFT's. Processor assignment is done either manually or by the task-assignment module. In other words, the application nodes are scheduled statically. The

problem of mapping nodes to processors is similar to that encountered by data-flow architectures.

The Lincoln architecture relies on special hardware to track the availability of data. This approach uses the Lincoln machine's hardware FIFO queues and the efficiency gains offered by processor locality. Figure 17 illustrates the design of a single TMS 32010 processor. Data-flow concepts could be simulated in the object code but this imposes a heavy communications overhead contrary to the real-time processing requirements of the system.

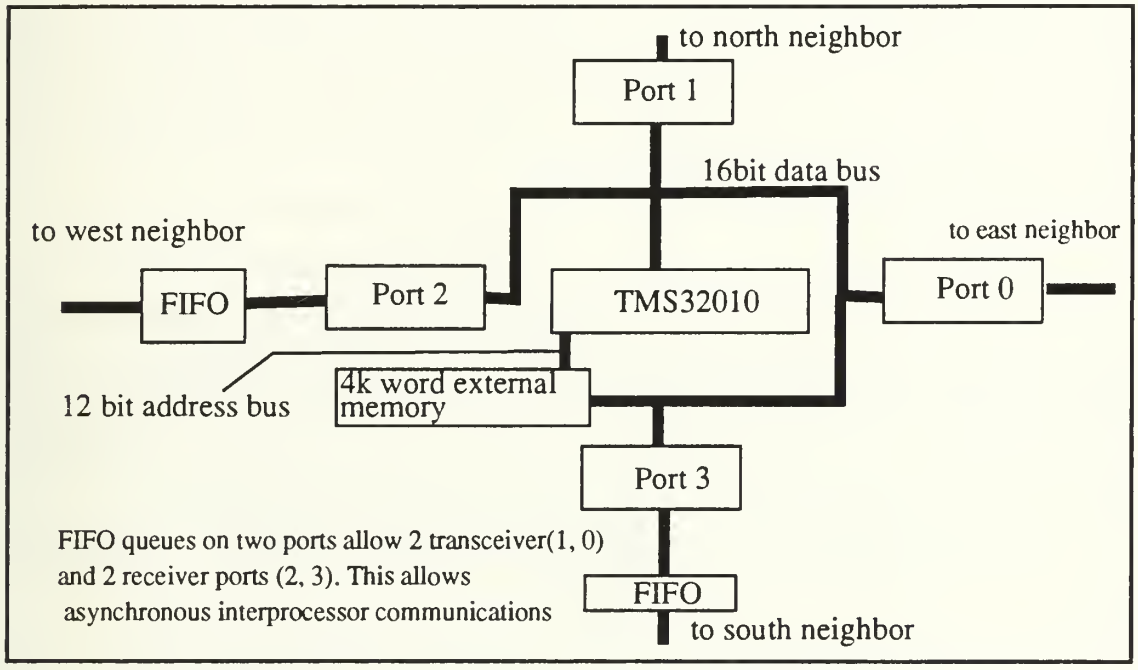


Figure 17: Texas Instrument TMS 32010 DSP [Ziss 87].

b. Entering/scheduling an application graph

Block Diagram Compilers are normally used as parts of simulation languages for digital signal processing. The Lincoln approach differs in two ways. First, it uses a graphic input interface to enter the application to the machine. The second difference is that instead of providing simulation code for a general purpose computer the compiler directly produces efficient object code to run in real-time on a MIMD array. When the

system schedules nodes statically it takes the physical arrangement of nodes and their processors into account. The compiler takes a graphical representation of a real-time DSP application and translates it into efficient assembly language code for each processor.

MIMD systems are often difficult to program as the programmer must

(1) partition the problem among the processors,

(2) route the interprocessor data transfers, and

(3) write different code for each processor in the array.

The system is designed to perform these three steps automatically. Signal processing problems usually have enough inherent structure to allow efficient mapping onto a MIMD array. The structure typically takes the form of parallelism and pipelining and is well represented by a directed graph. As a result the system can use an application's graph representation as high level compiler input.

c. Node assignment

Nodes assigned to the same processor are linked by common memory locations within the processor. I/O routines are created to transfer data between nodes in different processors. If the terminals of an interprocessor data transfer are assigned to adjacent processors, the routing is trivial. If the two processors are not adjacent, store-and-forward routines are generated for the intermediate processors, yielding a simple packet network.

The development of the compiler was eased by the choice of an asynchronous MIMD array hardware target. Because intercell data transfers are designed to be asynchronous the need for BDC software for insuring lock-step synchronous transfers between cells was obviated. Thus, the TMS32010 assembly code controlling I/0 transfers became simple to implement because hardware handles most of the data availability overhead.

3. Problem Partitioning and Task Assignment

Given a specification of the signal processing operations by a block diagram, the components of this specification, the nodes, must be assigned to individual processors in the array. At its simplest level, the structure of the array makes it possible for any block to be assigned to any processor and have the appropriate signal paths routed between processors.

a. Assessing Assignment quality

While this simplistic assignment strategy might suffice for uncomplicated situations it begs the question during high system utilization. Figure 19 illustrates the random assignment of the simple graph in Figure 18. In this case we see the high communications overhead if assignments are not chosen with respect to locality. Operator 1 is assigned to a random processor, as are the others. Communications from OP1 (heavy black arrows) traverse a circuitous path to get to OP2 and OP3. Results from OP2 (horizontal stripes) and OP3 (hashed arrows) then wend their way to OP4. Obviously, criteria need to be established and enforced to assess and then ensure the quality of each assignment.

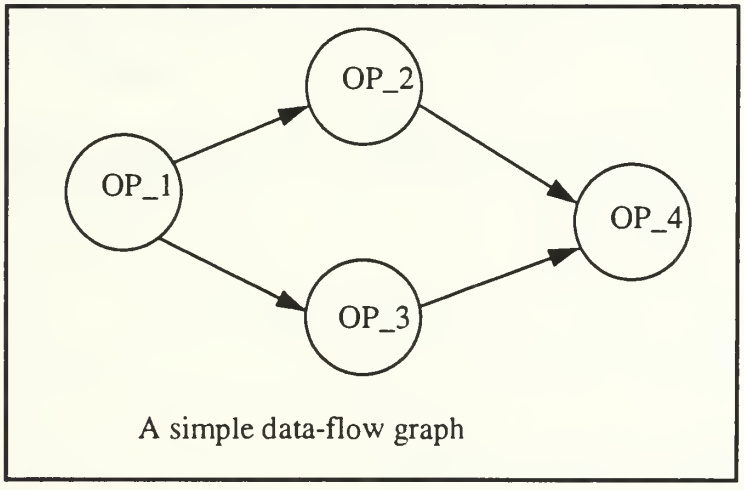


Figure 18: An example data-flow graph

For example, the lack of a global memory demands that the memory capacity of each processor may not be exceeded. An intuitively appealing criterion, as opposed to a constraint, is to minimize the number of processors used in the assignment. This global criterion is used to reduce the complexity and emphasize the conciseness of an assignment. These requirements must be taken into account both to make a reasonable assignment of nodes to processors and to assess the quality of the assignment.

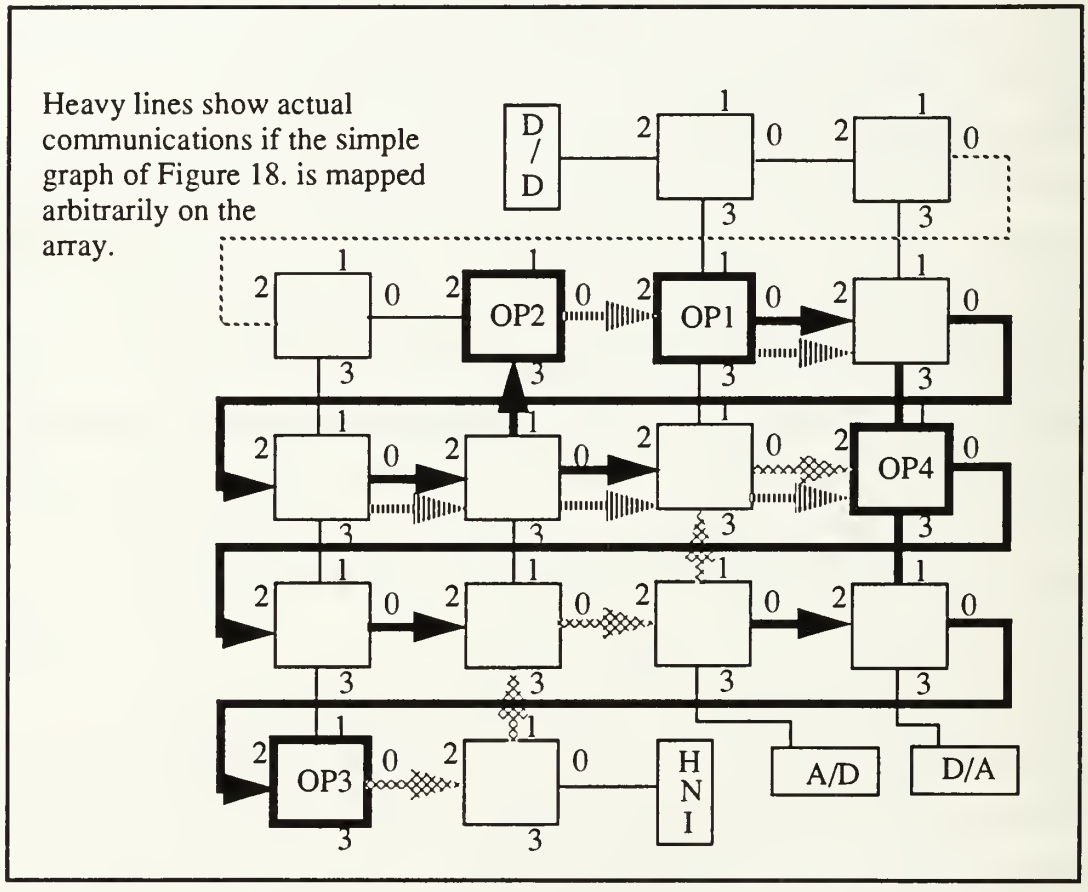


Figure 19: Arbitrary assignment of graph

b. Optimizing the Assignment

To achieve an assignment of signal processing components to computational processors that satisfied a set of both local and global criteria an

optimization problem was defined with a cost function which reflected these requirements. The independent variables over which the optimization was performed are the processor assignments for the nodes and signal routes through the array. These variables are fundamentally discrete; thus, optimization procedures that require the evaluation of a derivative could not be used. Instead, a combinatorial optimization procedure is necessary.

4. Algorithm Description

Simulated annealing was chosen because it answered the need for an optimized solution of discrete variables. It can be specified by identifying a set of solutions together with a cost function that applies a value to each solution. There exists an optimum solution which has the minimum cost possible. There may, of course, be more than one optimum solution. The Algorithm is the same as described in the previous section on CAPS with the exception that the grid architecture has different costs to optimize. The main local and global costs are summarized below:

a. Chosen Local Functions:

(1) Memory--The memory (Mreq) required for computations is evaluated for each processor. If this amount is less than 90% of the total available (Mavail) the cost function is zero. If greater than this number, the cost function equaled:

$$Mreq^{(-0.9 \times Mavail)^2}$$

As the TM532010 has separate program and data memories, the memory cost function was evaluated for each and summed.

(2) Real-Time--A cost function similar to that used in the memory usage was used to assess computational requirements. The number of cycles required by all of the blocks assigned to a processor were summed. If less than 90% of the total available time, the cost function is zero; if greater, the cost function assumes a quadratic form.

(3) Input/Output--In addition to the impact of signal computations on the memory and processing power of a single processor, the assignment of computational demand by the input/output programs required by the signal routing mechanisms is also made. The memory and computations required to support the routing are included in determining the memory and real-time components of the cost function.

(4) Special Capabilities--Several processors have special "capabilities" that distinguished them from the others. For example, only one processor had an A/D and D/A converter and another had the host-network interface. A subtle capability that is common to all processors is their presence. The processor array is assumed to be a rectangular grid, with some of the grid points having no processor. This capability allows the specification of no longer functioning processors and irregular geometries. Those blocks in the original block diagram requiring these capabilities are noted. If such a block is assigned to a processor lacking a specific capability, this component of the cost function is given a large non-zero constant.

b. Chosen Global Functions:

(1) The length of each signal route is measured in terms of the number of intervening processors. If this signal is not involved in a feedback loop, two times the length of the signal route is added to the cost function. If part of a feedback loop, ten times the length is added. This component of the cost function has the effect of reducing the number of processors used to support signal processing. Because of their inefficiency, feedback loops are especially penalized so that the components of each loop are kept physically close. If possible they are mapped to the same processor. Figure 20 illustrates a possible new assignment of a simple graph with these overheads taken into consideration.

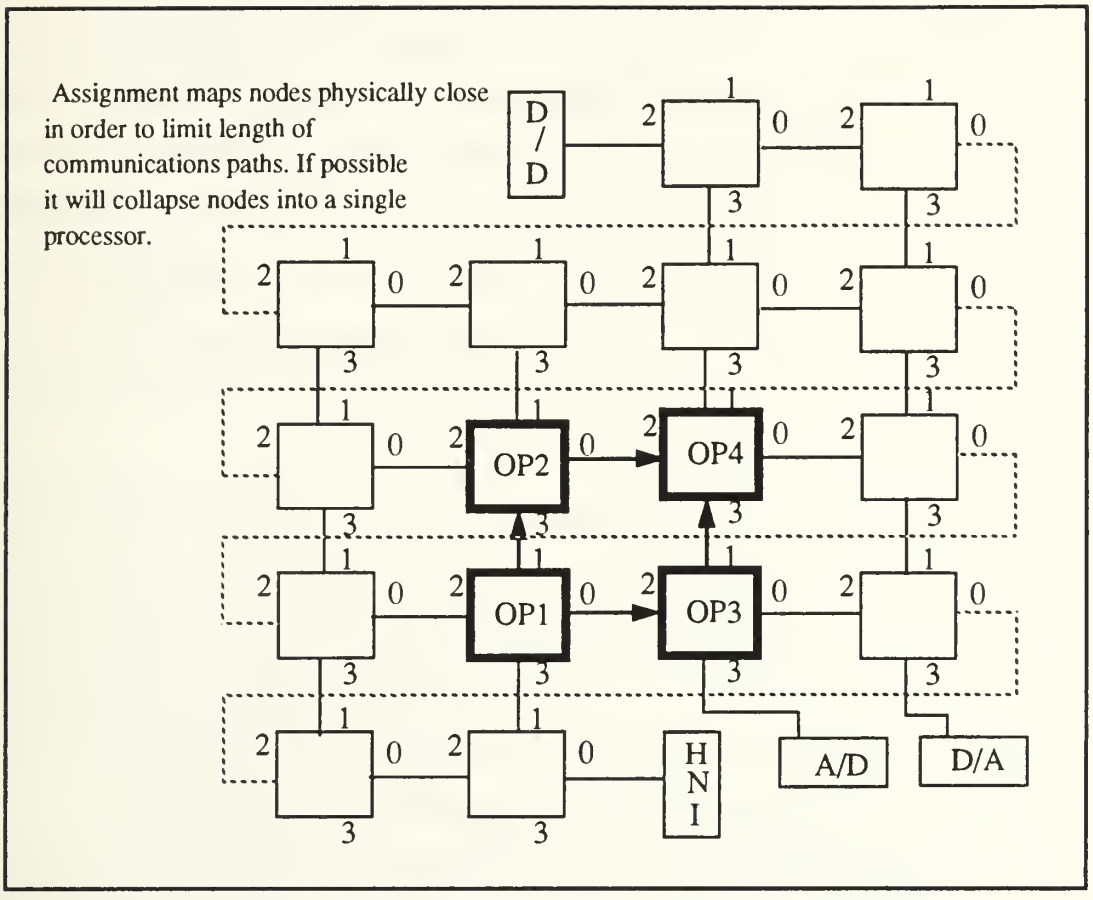


Figure 20: Optimized static assignment of nodes to processors

(2) In the context of simulated annealing a perturbation of the assignment of nodes and signal routing is made. With probability 1/4, a node is randomly assigned to another processor in the array and the attached signals rerouted. With probability 3/4, a signal is chosen randomly and a different routing for the signal made. The routing algorithm has probabilistic aspects as well. A small number of random routings between the two processors containing the signal routing components are made and the one having the smallest length chosen as the new routing. If a signal does not require interprocessor routing (i.e.: The nodes are assigned to the same processor) the intraprocessor routing is always chosen.

With these definitions of cost function and of what constitutes a random perturbation, the simulated annealing algorithm requires several thousand iterations to determine the optimal assignment. The "temperature" is reduced geometrically at each iteration (a reduction of about 0.9995 is used). The initial value of the temperature is equal to twice the maximum change of the cost function when ten random trial assignments are made: typically, this value is several hundred "degrees". The terminating threshold value for the temperature is fixed at 0.1. Although the minimum cost assignment is not always found, the real-time and memory constraints are always met. Typically, sub-optimum results have inefficient signal routes.

The intent of the BDC and the array is to bring a real-time environment to applications too large for a single processor, but without the detailed programming often required for parallel computation. Real-time performance is not obtained by assigning each node to its own processor and having a compiler determine an optimal signal routing but instead by having the program for each processor consist of tightly coupled, efficiently debugged program modules with a minimum of interprocessor computation.

MIMD architectures are more general than other multiprocessors. Despite their usual synchronization overheads they can be used to advantage with data-flow and large grain computation [Lewis, p.210]. The approach used in the TMS machine allows some asynchronous operation and so eases the control overhead faced in synchronous machines. There are other benefits as well. The use of a grid with specifiable processor degradation yields an architecture that fails more gracefully than a synchronous machine in the event of processor failure or system error.

The distributed memory of the architecture does impose global limits on the memory capacity of the machine and so limits its flexibility. Another shortcoming is that there is no code optimization for groups of programs chained onto a single processor. Nonetheless, The Lincoln machine gives us insight as to how a heuristic algorithm can be used to statically schedule a graph for real-time on a MIMD array. Further information can be found in [Ziss. 87].

C. Optimal Implementation of Flow Graphs on SSIMD Multiprocessors.

The next approach we discuss was developed by Barnwell and Schwarz [Barnwell 84] at the Georgia Institute of Technology. It is a general technique for the implementation of recursive and nonrecursive signal flow graphs and other arithmetic algorithms on synchronous digital machines composed of many identical programmable processors.

1. Optimality

Barnwell [Barnwell 84] defines three different categories of optimality: An implementation is said to be rate optimal if it achieves its sampling, or input rate, bound. It is delay optimal if it does not exceed its delay, or output rate, bound. Lastly, it is processor optimal if it exhibits perfect processor efficiency such that every cycle of every processor is used directly on the fundamental operations of the algorithm and no cycles are used for synchronization or systems control. These definitions are not mutually exclusive and any implementation could satisfy the criteria.

The Georgia Tech approach is characterized by two fundamental properties:

First, it uses the Skewed Single Instruction Multiple Data (SSIMD) mode in which exactly the same program is executed on all the processors, and that program is an exact single processor realization of the entire algorithm being implemented.

Second, all the data precedence relations among the processors are automatically maintained by the inherent synchrony of the system. This often results in processor-optimum solutions in which the use of M processors leads exactly to an M-fold increase in the system throughput.

These techniques result in a procedure in which the algorithm is specified in some simple notation, such as a set of difference equations, and from this a completely parallel multiprocessor implementation for the algorithm is generated.

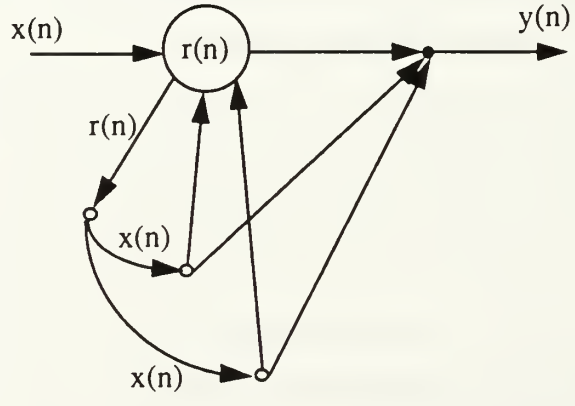
The resulting implementation is always either processor-optimum or time-optimum in which case the absolute throughput limit for the technique has been reached. In addition, for a large class of recursive signal flow graphs, the implementations are

absolutely optimum in the sense that there is no other implementation for a particular signal flow graph and a particular constituent processor. The approaches discussed here have been tested on a synchronous multiprocessor system.

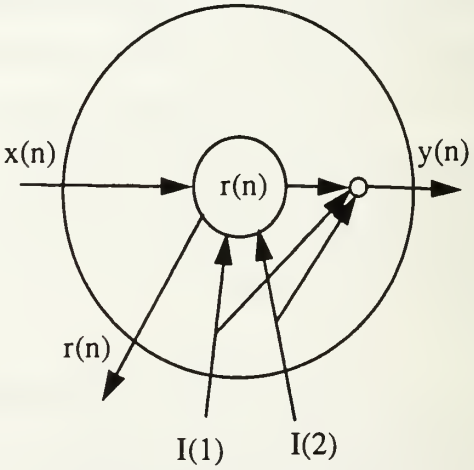
2. The SSIMD Mode

The fundamental computational mode which is utilized in these implementations is the Skewed Single Instruction Multiple Data Mode. In this mode, exactly the same instruction stream is executed on all processors, but with a fixed time skew maintained between the instruction execution times and the separate processors. The program realizes exactly one time-iteration of the flow graph. Figure 21 illustrates a Digital signal flow representation and a single processor realization of the same.

In a single processor realization, none of the delay elements are realized directly, but rather the output from each delay element becomes an input to the program and the input to each delay element becomes an output of the program. In the SSIMD realization, these delayed values are not computed by this processor, but are supplied from identical computations on other processors.



Signal flow graph for a second order recursive Direct form II digital filter



Single processor realization of the digital filter. All delays are not implemented by the program but are realized by the parallel structure

Figure 21: Recursive digital filter flow graph [Barnwell 82].

Figures 22 and 23 show a single processor and a two processor SSIMD realization for the signal flow graph of Figure 21. In the single processor solution of Figure 22, all of the past values of r(n) are supplied by the same processor, and there is never an issue of data availability. In the two processor realization of Figure 23, alternate points are supplied by each processor, and the two processors must be skewed such that the data requirements of each is always met by the other.

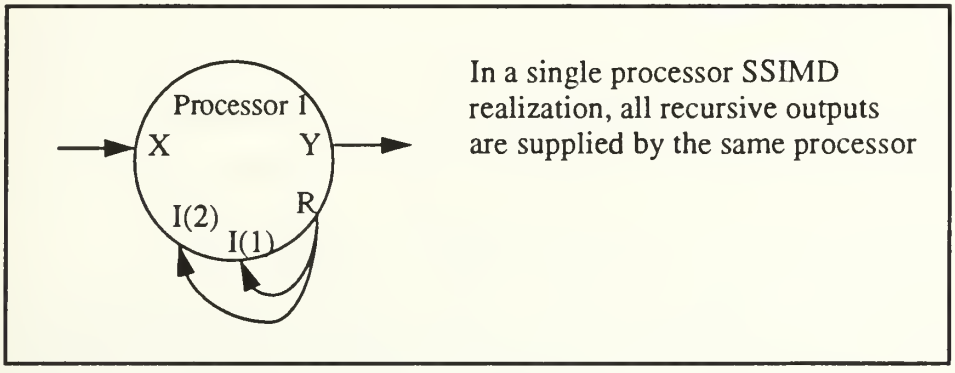


Figure 22: SSIMD single processor realization of a recursive filter [Barnwell 82].

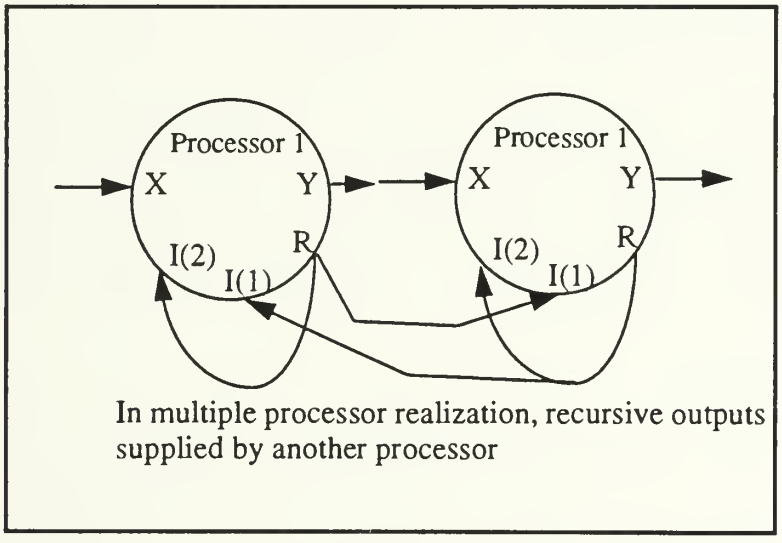


Figure 23: Multiple processor realization of the recursive filter of Figure 21 [Barnwell 82].

It should be noted that these SSIMD solutions are "free running" such that whenever a processor completes the computations associated with one time index, it immediately begins the computations associated with another time index. Hence each program realizes an infinite loop and, under the assumption that the program timings are not data dependent, each loop takes exactly the same amount of time to execute. Thus, if M processors are started at times t(m), 0<m<M-1, then the relative time skews so imposed remain fixed until the programs are halted externally.

The problem of implementing a particular iterative arithmetic program reduces to specifying the M starting times, t(0)...t(m-1), such that all the data available for the various computations is available before it is needed.

3. Implementing Recursive Arithmetic Programs

The problem of implementing a particular recursive signal flow graph in SSIMD mode can be divided into two related problems. The first is the problem of finding and characterizing all legal SSIMD solutions for a particular single processor program for implementing the signal flow graph. The second problem is that of constructing the particular single processor program such that the eventual SSIMD scheduling solution will be optimum. This section addresses the first problem for single input/single output signal flow graphs. These results are easily extended to multiple input/output systems.

In fitting the programs together in SSIMD mode, the data which must be used include the length of the program, T, the times at which the delayed recursive inputs are first used, I(L)...I(l), for a system with longest delay and the time at which the recursive output is available, R.

The first point to note is that all SSIMD solutions are bounded by the solution with equally spaced starting times. It can be proven that in SSIMD, the processors operate in a circular fashion, and the relationships between a single processor and its predecessors and successors in the processor chain are identical for all processors in the system. Any

advantage attained by local time perturbations at one processor would be lost at some other processor. Hence, all SSIMD solutions are bound by equally skewed solutions.

Based on these results, four important features should be noted.

First, given a single processor program for a signal flow graph (or other algorithms describable in a similar fashion), the maximum number of processors which can be used is immediately available and the starting times for the processors in SSIMD mode are simply computed. Hence, for a given program, the SSIMD implementation procedure is very simple.

Secondly, and more importantly, the maximum number of processors which can be used to advantage is a function of a single time index, $I(l(x))$, $1 < l < L$, where L is the longest delay in the system. Hence, a simple constraint exists for optimizing a particular program for SSIMD implementation. The program is obtained by maximizing the minimum number of processors, $M(l)$ which could be utilized on any arbitrary recursive input whose time of delay was the constraint on the system [Barnwell 82].

Third, and perhaps most importantly, the optimum time skew is not a function of either the program duration or the number of recursive inputs or outputs of the program. This allows for several important generalizations to be made and, for properly written programs, leads to impressive solutions. For example, the system of Figure 21 can typically be implemented with 8 or 9 processors even though it has only two recursive inputs. The throughput gains for a data-flow architecture working with recursion are immediate.

Finally, it should be noted that there are no constraints at all if the algorithm is reconcursive. In a theoretical sense, this is a trivial statement, since it is clear that if there are no constraints on data availability, then any number of processors can be used to advantage. However, in an implementation sense the SSIMD approach still leads to elegant processor-optimum solutions for any number of processors.

4. Optimum Signal Flow Graph Implementation

A study produced a set of systematic procedures for generating single processor programs which could produce optimum realization when utilized in the SSIMD mode. The problem addressed was how to proceed in an automated fashion from a simple representation of a signal flow graph, such as a set of difference equations or a matrix representation, to a single processor program which maximized the minimum value of M(l). This solution can be found by systematically investigating both the computational orderings of and at the nodes. It is easy to see that it can be accomplished inefficiently by an exhaustive search.

The most important result, however, concerns the optimality of the SSIMD solutions. For a very large class of signal flow graphs, including both the normal and transposed forms of all direct form, cascade, and parallel digital filters, the SSIMD solution is absolute optimum in the sense that, for a particular constituent processor, it achieves the greatest possible throughput for the fewest possible processors.

This can be illustrated in the context of the example of Figure 20. First note that in order maximize the number of processors used the quantity needed to make recursive feedback available must be minimized. This requires that each of the recursive delayed inputs, I(1) and I(2) in Figure 22b, be first used as near in time to the completion of the computation of the recursive output, R, as possible. This leads to the general principal that when ordering the computations at a node, the delayed recursive inputs should be used last. This shows that the system throughput is not a function of the length of the program or the number of delayed inputs, but is only a function of the input/output time for one result and the time of one multiply/add operation. These are fundamental constraints of the processors themselves.

Further, the output/input of a result and the multiply/add operations are the minimum possible required computations in a recursive signal flow graph. Since a single processor realization involves the fewest possible special (non-arithmetic) operations, it

also achieves this throughput with the fewest processors. These results can be generalized for a large class of recursive signal flow graphs, which lead to several important points.

The first is that the SSIMD implementations are generally simpler than other multiprocessor options which typically include the parsing of the signal flow graph to promote local parallelism. By including everything needed for each instantiation of an application graph on each of the processors the overhead of interprocessor communications is minimized. This requires individual processors capable of handling the entire graph.

The second important point is that all the limits on the number of processors and the throughput are a reflection of the recursive nature of the programs. As previously noted, if there is no recursion, then the solution is no longer constrained by the algorithm but rather by the nature of the hardware.

The largest potential problem in SSIMD solutions concerns the interprocessor communication issues. Since the entire SSIMD development is done under the assumption that the processors can communicate "at will", this would first appear to be a critical issue. It turns out, however, that it is not. This is true for two reasons.

First, the fundamental periodicity of the SSIMD solution makes the communications requirements very uniform, which avoids many potential time conflicts. second, and most important, the nature of the communications environment can be systematically controlled. To see this, one simply needs to note that the number of processors with which a particular processor must communicate is controlled by the maximum length of the delay elements in the application graph.

The use of long delay chains does improve the final solution since it leads to SSIMD realizations which require fewer processors to realize a time-optimum solution. But the entire procedure still works if the maximum delay length is constrained to be one. This is the case for the classical formulation for signal flow graphs. For such realizations, each processor only communicates with its two nearest neighbors, and communications are always unidirectional. Such realizations have the same maximum throughput rate, but, in

general, require more processors to achieve it. Most important, however, they have a communications environment which is always trivially implementable.

SSIMD first fully distributes the algorithm in time because a separate time index is assigned to each processor. It then explicitly maps this time distribution into space. The difference between this and a systolic array is that a systolic implementation only maps the algorithm in space. The SSIMD approach is more processor and rate optimal than a systolic array. The primary advantage of SSIMD comes from the fact that instead of viewing the problem from the system clock, time is referenced at the individual processors and so a complex timing problem in the systolic array becomes a relatively simple one in the SSIMD paradigm. More information on the approach is available in [Barnwell 82] and [Barnwell 84].

D. ATAMM: A Paradigm for Predictable Performance in Real-Time on Multiprocessors

1. The Algorithm To Architecture Mapping Model (ATAMM)

Som, Mielke, and Stoughton of Old Dominion University are working on the development of strategies for predictable performance in homogeneous multicomputer data-flow architectures operating in real-time [Som 90]. The approach is restricted to large-grained, decision-free applications such as the real-time implementation of control and signal processing algorithms. The mapping of such algorithms onto data-flow architectures is realized by a new marked graph model called ATAMM (Algorithm To Architecture Mapping Model). Algorithm performance and resource needs are determined for predictable periodic execution of algorithms. Predictability in performance and resource requirements is achieved by algorithm modification and input data injection control. Performance robustness is gracefully degraded to adapt in the event of decreasing numbers of resources. Two key areas the approach focuses on are as follows.

First, the ability to achieve predictability of algorithm performance is considered to be the most important feature of real-time computing. It sometime is more important than

the actual performance. The design of such a system must have precise knowledge about the time of input arrival and output generation for the algorithm, not simply knowledge of statistics concerning average performance. However, predicting algorithm performance accurately in multicomputer data-flow architectures is known to be very difficult as most scheduling problems in a multicomputer environment are NP hard.

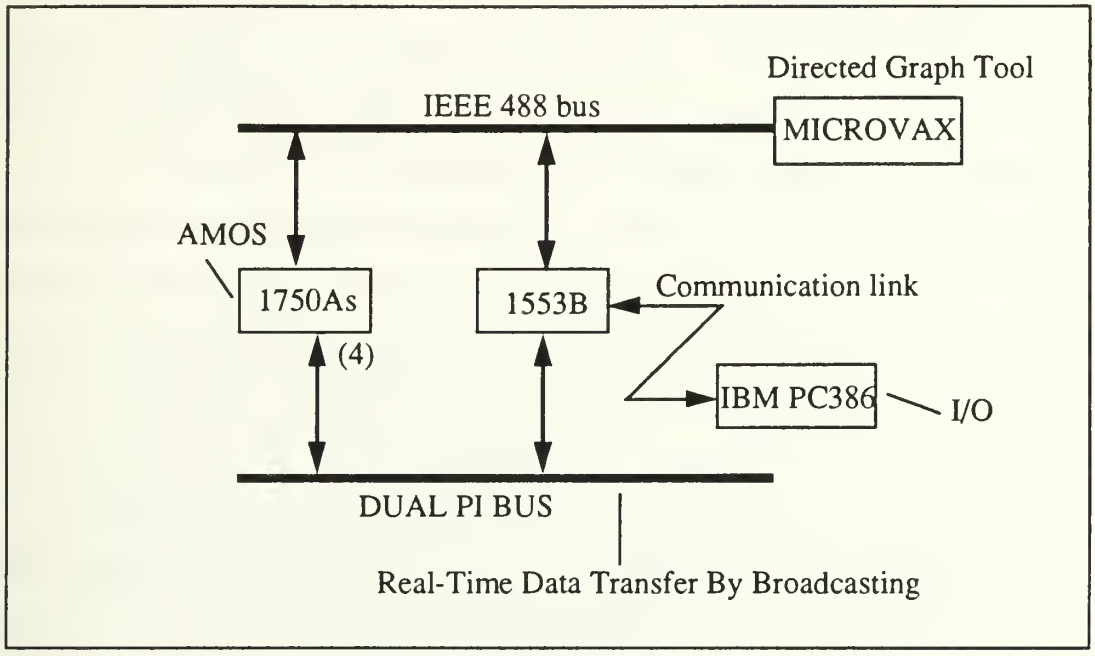


Figure 24: ATAMM Architecture [Som 90].

Second, very little research has been directed towards resource management in data-driven computing. The execution of algorithms must be controlled so that resource need does not exceed resource availability. Otherwise data packets experience unnecessary waiting times and require extra storage space, and system performance becomes unpredictable.

This scenario is unacceptable in real-time computing with hard deadlines for outputs. New abstract computational models are required for real-time data driven computations which lead to algorithm performance and resource requirements that are predictable. On going research at Old Dominion University has led to the development of

a new marked graph model for describing the execution of algorithms in real-time data-flow architectures, ATAMM.

The model specifies the criteria for a multicomputer operating system to achieve predictable performance within resource constraints. It represents the applications as directed acyclic graphs. The architecture is assumed to consist of two to twenty identical functional units or resources each having a capability of processing, communication, and memory. The number of algorithm nodes in a problem is not expected to be more than twenty. This range of functional units and algorithm nodes is selected due to the large-grained aspect of the target algorithms and knowledge of target architectures.

The approach to achieving predictability in performance and resource requirements is to modify the algorithm graph and to control the input data injection rate so that a functional unit is always available for every enabled algorithm node. Algorithm performance is characterized by throughput and computing speed. When sufficient resources an available, the system executes algorithms with maximum throughput and maximum computing speed and the corresponding resource requirement is predicted.

The programmer can develop strategies for graceful degradation in performance when only limited resources are available or when resources fail. The user is able to specify, off-line, trade-offs between decreasing throughput or decreasing computing speed with the help of a software design tool. The operating system is able to implement these changes on-line in real-time as the number of resources decreases.

2. The ATAMM Model [Som 90]

ATAMM describes algorithm execution on a data-flow architecture by three marked graphs, the algorithm marked graph (AMG), which is similar to the input graph used in RC scheduling, the node marked graph (NMG), and the computational marked graph (CMG).

a. Algorithm Marked Graph

An algorithm marked graph is a marked graph which represents a decomposed algorithm. Transitions and places represent algorithm operations and operands respectively. The transition times represent the computational times required for the algorithm operations. The algorithm marked graph contains an edge (i, j) directed from node i to node j if the output of node i is an input for node j. Edge (i, j) is marked with a token if the output from node i is available as an input to node j. All edges can have a queue and accommodate more than one token at a time.

To illustrate the representation of a computational problem consider the directed graph in Figure 25. This input graph is used by ATAMM and is similar to that used in RC analysis.

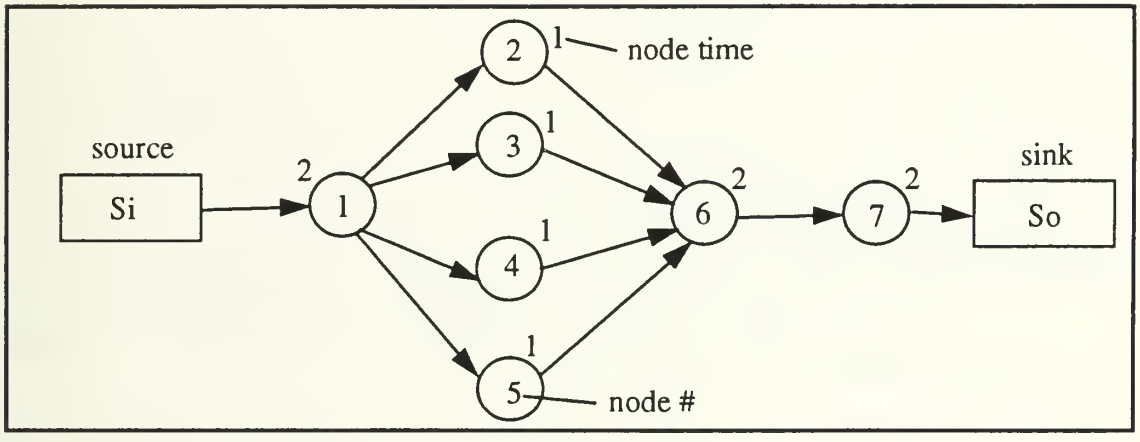


Figure 25: ATAMM input graph (AMG) [Som 90]

b. Node Marked Graph

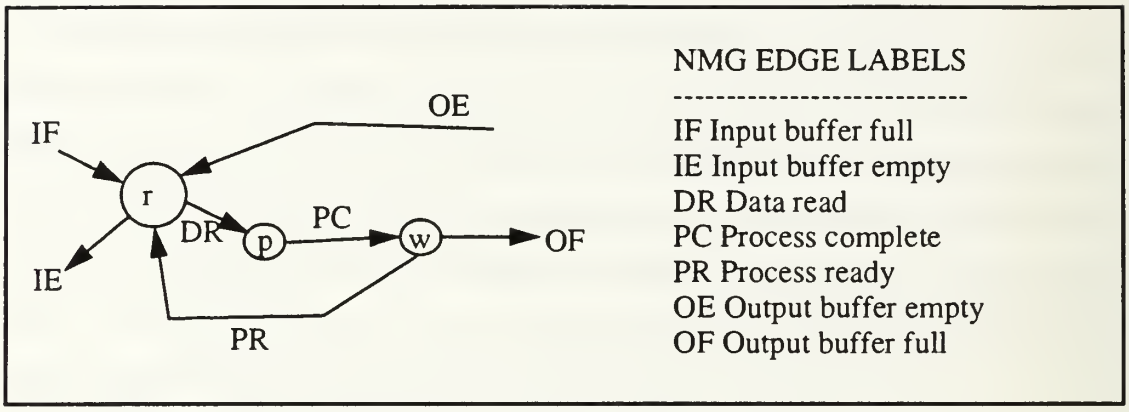


Figure 26: A sample Node Marked Graph [Som 90].

The node marked graph (NMG) is a representation of the execution of a transition on the AMG by a functional unit. Three primary activities, reading of input data from memory (r), processing of input data to compute output data (p), and writing of output data to memory (w), are represented as transitions in the NMG. Data and control flow paths are represented as places, and the presence of signals is notated by tokens marking appropriate places. The NMG for an AMG transition is shown in Figure 26.

The conditions for firing the process and write transitions of the NMG are as defined for a general Petri net, while the read transition has one additional condition for firing. A functional unit must be available for assignment to the algorithm operation before the read node can fire. Once assigned, the functional unit is used to implement the read, process, and write operations before being returned to a queue of available functional units. The initial marking for a NMG consists of a single token in the process ready place so that only one functional unit can work on an AMG transition at a time (static data-flow architecture). However, the Output Buffer Empty (OE) edge may contain a number of tokens so that the execution of an AMG transition can be repeated by another functional unit before the output is consumed. The total initial number of tokens on OE and OF edges is the size of the output queue in edge OF.

c. Computational Marked Graph

The computational marked graph (CMG) is constructed from the AMG by replacing every transition by the corresponding NMG. AMG places are replaced by place pairs, a forward directed place representing data-flow and a backward directed place representing control flow. The performance measure TBIO (time between input and output) is the elapsed computing time between an algorithm input and the corresponding algorithm output. Therefore, TBIO is an indicator of computing speed. The algorithm-imposed lower bound for TBIO, denoted TBIO(lb), is given by the sum of transition times for nodes contained in the longest directed path (critical path) from the input source to the output sink in the AMG.

The performance measure TBO (time between outputs) is the elapsed computing time between successive algorithm outputs when the AMG is operating periodically at steady-state. Therefore, the inverse of TBO is an indication of output per unit time or throughput. The algorithm imposed lower bound for TBO (TBO(lb)) is given by the largest time per token of all directed circuits in the CMG. A second bound on TBO is imposed by the availability of resources. The resource-imposed minimum value of TBO is given by TCE/R where TCE (total computing effort) is the summation of all the transition times of the AMG and R is the number of available processors.

3. Injection Control

When presented with continuously available input data packets, the natural behavior of a data-flow architecture results in operation where new data packets are accepted as rapidly as the available resources and the input node of the AMG permit. This leads to operation at a steady-state where TBIO > TBIO(lb). This occurs because the pipeline from input to output becomes congested with extra data packets which must wait for free resources to be processed. From bounds on TBO, the output of the AMG cannot be generated at a rate higher than 1/TBO(lb) or R/TCE. Injection control is a control procedure

which limits the maximum rate at which new input data packets can be injected from the source. Therefore, injection control eliminates data packet congestion and thus preserves operation at TBIO(1b).

Two diagrams which display graph play and are useful for determining the number of resources needed to achieve specified performance measures are the SPG and TGP. The SGP (Single Graph Play) diagram displays the execution of each node of the AMG as a function of time. The diagram is constructed for a single input data packet assuming availability of a resource for every enabled node. When several nodes are active at the same time, lines indicating node activity are stacked vertically so that computing concurrency is apparent. a sample SGP diagram shown in Figure 27.

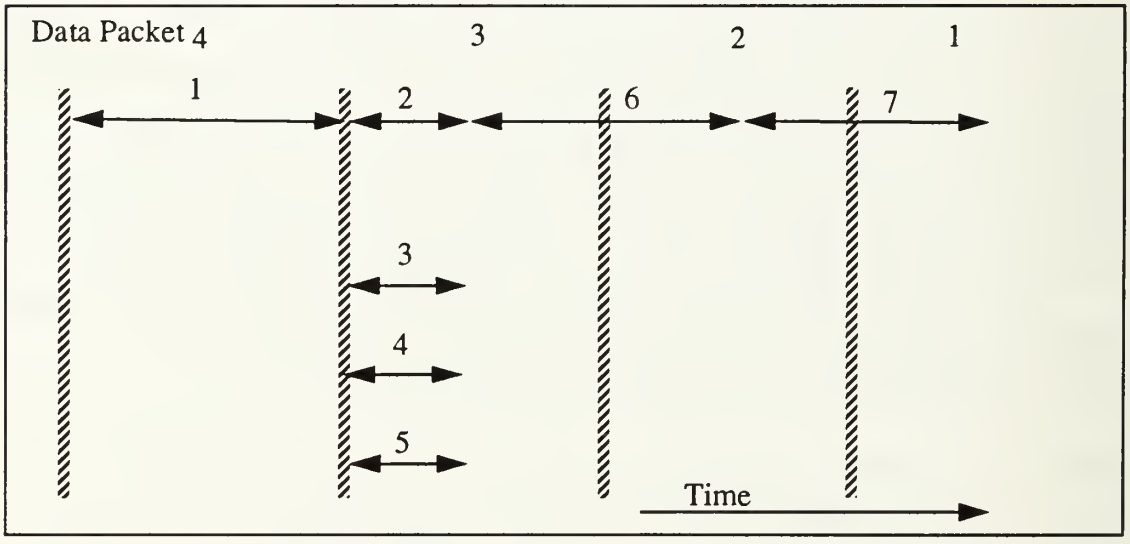


Figure 27: Simple Graph Play diagram for the graph of Figure 25 [Som 90].

The resource requirements to execute a single data packet are obtained by counting the number of active nodes during each time interval in the SGP diagram. The peak resource requirement is denoted by Rmin and represents the minimum number of resources required to achieve SGP. As an example, Rmin is 4 for the graph of Figure 24.

The TGP (total graph play) diagram is a diagram which displays the execution of each algorithm node when the algorithm is executed periodically in steady-state with

period TBO. As with SGP, the diagram is constructed under the assumption that a resource is available for every enabled node. The TGP diagram is drawn using information from the SGP. SGP is divided into segments of width TBO and these segments are overlaid to form TGP. Each segment from SGP represents a new input data packet. Data packets are numbered sequentially so that the packet numbered (i+1) is the data packet which is input to the algorithm TBO time units after the packet numbered i. To illustrate the construction of this diagram, TGP for the graph of Figure 24 is shown in Figure 27.

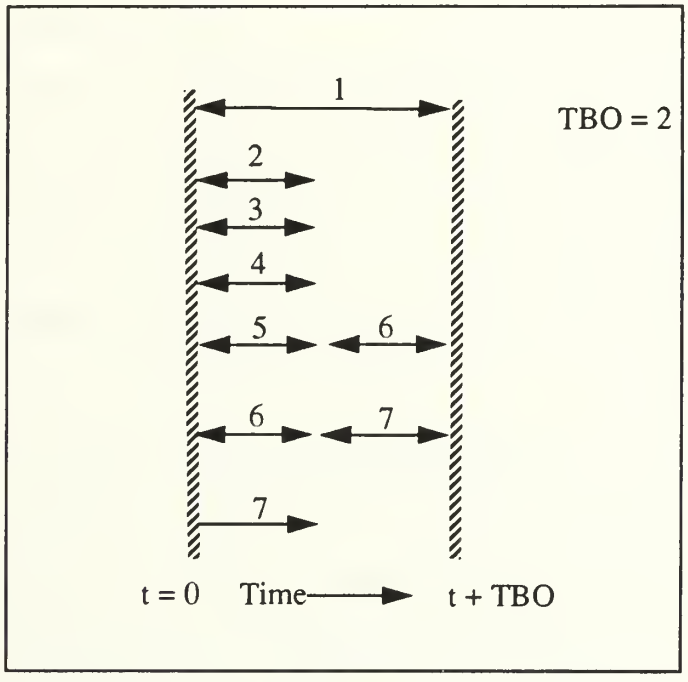


Figure 28: Total Graph Play of Figure 25's graph [Som 90].

The resource requirements to execute multiple data packets injected with period TBO are obtained by counting the number of active nodes during each time interval in the TGP diagram. As TGP is periodic at steady state with period TBO, so also is the total resource requirement. The peak resource requirement necessary to execute the graph periodically with period greater than or equal to TBO is denoted Rmax.

Rmax is determined by finding the largest resource requirement in all TGP diagrams drawn for injection intervals greater than or equal to TB0. For example, the TGP

diagram drawn for TB0 = TBOlb = 2 shown in Figure 27 indicates that a minimum of 7 resources is required. If this same TGP is drawn for all values of TBO = 2, it can be shown that the required number of resources does not exceed 7.

V. CONCLUSION

A chinese proverb says "to know the road ahead, ask those coming back". The approaches of the previous chapter give an insight into the relative merits and possible shortcomings of both the AN/UYS-2 and RC analysis. Through the perspective of other real-time systems an insight is gained into furtherance of the system's performance possibilities.

A. The RC Approach in Context

Revolving Cylinder analysis is a flexible policy developed to improve the performance of the AN/UYS-2. It is unusual because it actually takes control and communications overheads into consideration when executing in real-time. Its attraction lies in its ability to reduce these overheads in the system while maintaining the fullest possible utilization of all processors. RC analysis can be implemented on a variety of architectures and has merit beyond the confines of the AN/UYS-2 architecture.

1. Static vs. Dynamic Node - Processor Assignment

The AN/UYS-2 schedules its nodes statically but allows the hardware scheduler to actually assign the nodes to a processor dynamically at run time. This keeps structure in the execution order of an application graph without introducing control overheads at run-time. RC scheduling gives a deterministic output flow rate with the caveat that the application's nodes must have a regular (i.e., non-branching) execution profile. The trade-off is that the system cannot guarantee that determinism because of a lack of prior knowledge about where the system will execute each particular node of an application.

CAPS uses a fully static scheduling approach to schedule and map execution nodes to a conventional processor. The use of the harmonic block allows the target machine to run a number of processes at different execution rates while still meeting real-time deadlines. RC scheduling in its current incarnation is rate-monotonic. The range of applications

suitable for the AN/UYS-2 can be increased by the addition of a flexible rate mechanism along the lines of CAPS.

Static assignment such as that found in the Lincoln and Georgia Tech machines will increase the determinism of the machine's output flow by inducing "lock-step" execution of each node. The AN/UYS-2 cannot implement this scheme without incurring huge communication penalties because of the common bus each resource uses to communicate with the scheduler and other processors/global memories.

2. Throughput During High Demand Periods

The performance of the AN/UYS-2 is improved under high loads with the implementation of the Revolving Cylinder [Akin 93] because of the increased determinism in throughput rates. The ATAMM approach seeks to control determinism through the control of data injection rates. While this does help induce regularity it loses some of the structure of the original data. This matters in the threat environment in which the AN/UYS-2 is going to operate.

The CAPS implementation suffers throughput degradation under high loads because slack is removed from the harmonic block and any kernel calls made will delay the execution of real-time processes past their deadlines. The inability to predict this delay through anything but statistical analysis is concerning in a real-time environment.

SSIMD can achieve high throughputs but the Georgia Tech machine is more suited to problems of finer granularity than those handled on the AN/UYS2 because it loads an entire application onto each processor. The execution of small application graphs is faster on the machine because of the inter-processor communications but the architecture is not as flexible as that of the AN/UYS-2.

The MIMD hardware of the Lincoln machine allows a locality of assignment not possible with the AN/UYS-2. The fact that the processors can communicate with each other without having to get on a common bus makes this an attractive idea. The ability to do this

reduces the non-determinism of output flow and improves throughput under high demand by dynamically assigning nodes to processors by proximity as well as availability.

3. Determinism of Output Flow Rate

The AN/UYS-2 implementation of RC scheduling produces output flow with a determinism that is dependent on the application graph's execution profile. If the execution graphs are inherently non-deterministic due to branching, recursion, etc... then the system's output flow reflects it. The SSIMD array can handle recursion smoothly by having one iteration of the recursion running on a processor fed into the next processor for another execution. This is not possible on the AN/UYS-2. The implied communications overhead burdens the data bus to the point where throughput is seriously degraded.

Input injection rate control is the method that ATAMM uses to induce regularity in its output flow. This approach can improve the regularity of the AN/UYS-2 but the arbitrary loss of data is unacceptable. There may be ways to implement this of approach without specifically controlling the injection rate. This involves the system keeping current input on hand in a read buffer. As the input changes, the value of the buffer changes, but there is always current data on hand for the start of a new graph instance.

B. Summary

RC scheduling addresses the determinism of the response time of a data flow machine. Other research in the field of data-flow machines used in real-time environments, with the exception of Old Dominion, note the importance of such determinism and then either ignore the problem or use statistical profiles of an application to build in a response cushion.

There are trade-offs in the approach insofar as deterministic execution profiles are required to produce deterministic output flows. More deterministic performance can be obtained from fully static scheduling policies but the RC approach offers a hybrid with the

flexibility and robustness of dynamic data-flow with some of the determinism and throughput performance of control-flow execution of each node.

C. Possible Improvements to The AN/UYS-2

The set-up, execution, and breakdown of nodes is one of the bigger overheads in the implementation of the RC schedule. Lee [Lee 90] addresses the concept of hardware implementation of functions normally performed through software. The main advantage of the approach is that each fetch, set-up, breakdown, and write is much faster if performed in hardware. This also enables processors to access shared memory the same way they access local memory.

The addition of nearest-neighbor communication paths between the AP's might allow more deterministic flow without the high overheads of a fully synchronous implementation. This parallels some of the ideas of the Lincoln labs machine without major alterations to the System hardware.

D. Future Research

The similarities of the Lincoln and Old Dominion machines to the AN/UYS-2 indicate that performance and throughput determinism gains are most easily found by mixing the balance of static and dynamic node scheduling. The RC technique is extremely good at wrenching deterministic output flow from an existing architecture without expensive modifications. These other approaches suggest that some gains can come from hardware changes and some few from software.

The investigation of interprocessor communications and the modification of data arrival rates are two promising avenues for further improvement of the AN/UYS-2 and the RC technique. Each of these are implementable at low cost and have the potential to increase the system's performance.

Another avenue of investigation is the CAPS system's use of multiple rate execution times. This capability can add flexibility to the range of applications the AN/UYS-2 can handle and increase the life-span of the system for years to come.

LIST OF REFERENCES

[Akin, 93]

Akin, C., *Efficient Scheduling of Real-Time Compute-Intensive Periodic Graphs on Large Grain Data Flow Multiprocessors*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1993.

[Barnwell, 82]

Barnwell, T. P., Hodges, C.J.M., and Randolph, M., "Optimum Implementation of Single Time Index Signal Flow Graphs on Synchronous Multiprocessors," *Proceedings, 7th International Conference on Acoustics, Speech, and Signal Processors* (May 82) Paris, France, pp. 679-682.

[Barnwell, 84]

Barnwell, T. P., and Schwarz, D. A., "Optimal Implementation of Flow Graphs on Synchronous Multiprocessors," *Proceedings, 17th Asilomar Conference on Circuits, Systems, and Computers* (Oct 83) pp. 188-193.

[Bell, 92]

Bell, H. A., *A Compile Time Approach for Chaining and Execution Control in the AN/UYS-2 Parallel Signal Processing Architecture*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1992.

[Gaudiot 87]

Gaudiot, J. L., "Data-Driven Multicomputers in Digital Signal Processing," *IEEE Proceedings*, 75, 9 (Sep 87), 1220-1234.

[Lee 87]

Lee, E. A., and Messerschmitt, D. G., "Static Scheduling of Synchronous data-flow graphs for Digital Signal Processing," *IEEE Proceedings*, 75, 9 (Sep 87), 1235-1245.

[Lee 90]

Lee, E. A., and Bier, J. C., "Architectures for Statically Scheduled Data-Flow," *Journal of Parallel and Distributed Computing*, v. 10, pp. 333 - 348, December 1990.

[Levine, 91]

Levine, J. E., *An Efficient Heuristic Scheduler for Hard Real-Time Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1991.

[Lewis]

Lewis, T. G., and El-Rewini, H., *Introduction to Parallel Computing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Little 91]
Little, B. S., *A Technique for Predictable Real-Time Execution in the AN/UYS-2 Parallel Signal Processing Architecture*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1991.

[Meng 91]
Meng, T.H.Y., Broderson, R. W., and Messerschmitt, D. G., "Asynchronous Design for Programmable Digital Signal Processors," *IEEE Transactions on Signal Processing* v.39 (Apr 91), 939-952.

[Rice 90]
Rice, M. L., "The Navy's New Standard Digital Signal Processor: The AN/UYS-2," paper presented at the Association of Scientists and Engineers 27th Annual Technical Symposium, 23 May 1990.

[Shukla 92]
Shukla, S. B., Little, B.S., and Zaky, A., "A Compile-Time Technique for Controlling Real-Time Execution of Task-Level Data-Flow Graphs", presented at the 1992 International Conference on Parallel Processing.

[Som 90]
Som, S., Mielke, R. R., and Stoughton, J. W., "Strategies for Predictability in Real-Time Data-Flow Architectures", *Proceedings, 11th Real-Time Systems Symposium,* pp. 226 - 235, IEEE Computer Society Press, Los Alamitos, Ca., 1990.

[Ziss 87]
Zissman, M. A., O'Leary, G. C., and Johnson, D. H., "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer," *Proceedings, International Conference on Acoustics, Speech, and Signal Processing*, v.4, (Apr 87)Dallas Tx, pp. 1867-1870.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexanderia, VA 22304-6145	1
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr. Amr M. Zaky Code CS/Za Associate Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	3
5.	CPT John H. Quigg 3070 Brookview dr. Marietta, GA 30067	3